A JavaScript Information Flow Monitor for Symbolic Testing

André Baptista Neves Ribeiro

Thesis to obtain the Master of Science Degree in Telecommunications and Informatics Engineering

Supervisors: Prof. José Faustino Fragoso Femenin dos Santos
             Prof. Ana Gualdina Almeida Matos

Examination Committee

Chairperson: Prof. Rui Jorge Morais Tomaz Valadas
Supervisor: Prof. José Faustino Fragoso Femenin dos Santos
Member of the Committee: Prof. António Paulo Teles de Menezes Correia Leitão

November 2019
Dedicated to someone special...
Acknowledgments

My first word of gratitude goes to my supervisors Ana Almeida Matos and José Fragoso Santos, who were always available to guide me towards the right path. I would also like to thank my friends, specially; João Lopes, Beatriz Abecasis, João Matos, Sergio Correia and Rodrigo Rocha for their feedback and motivation.

I would also like to thank the institutions responsible for the funding of this thesis as well as the host institution Instituto de Telecomunicações for providing the infrastructure for conducting my work. This work is partially funded by FCT within project Elven POCI-01-0145-FEDER-016844, Project 9471 - Reforçar a Investigação, o Desenvolvimento Tecnológico e a Inovação (Project 9471-RIDTI) and by Fundo Comunitário Europeu FEDER.

Finally, I would like to thank my family, who have always supported me.
Resumo

JavaScript é uma das linguagens mais relevantes para aplicações do lado do cliente, como pode ser visto pelo número de repositórios no github (http://githut.info). Antes limitado à validação de formulários e outras pequenas tarefas, hoje evoluiu para um componente importante da Web. O JavaScript é uma linguagem muito dinâmica, permitindo a criação de variáveis e funções em tempo de execução. No entanto, essa natureza dinâmica é uma faca de dois gumes, pois torna a análise uma tarefa complexa. As ferramentas atuais para análise de JavaScript são limitadas, principalmente devido à combinação do dinamismo mencionado acima (forçando as ferramentas a analisar pequenos subconjuntos da linguagem) e a heterogeneidade dos browsers (que interpretam o JavaScript de maneira diferente). Esta falta de ferramentas de análise torna o desenvolvimento de código JavaScript seguro um desafio não trivial.

Nesta tese, propomos uma solução baseada em técnicas de análise de fluxo de informação, que fornece fortes garantias independentemente do browser. Apresentamos uma ferramenta que consiste num monitor de fluxo de informações, que visa manter a confidencialidade, capturando fugas de informação tanto explícitas como implícitas. Este monitor analisa a linguagem intermédia JSIL, uma linguagem simples criada para a verificação de JavaScript. Para que um programa seja analisado, ele precisa ser compilado no JSIL através do compilador JS-2-JSIL, que foi extendido para conter comandos que suportam monitores. Estes novos comandos permitem-nos anotar variáveis com níveis de segurança e manipular essas anotações. Por fim, aproveitamos o fato de que JSIL suporta a execução simbólica para analisar o programa percorrendo vários caminhos de execução.

Propomos um monitor que visa manter a confidencialidade, capturando fugas de informações tanto explícitas como implícitas. Para isso, definimos formalmente a sintaxe e a semântica das regras que o monitor segue. De seguida implentamos essas mesmas regra. Finalmente, avaliamos a execução através de um conjunto de testes unitários para avaliar o monitor, provando que podemos detectar em JSIL todos os tipos de fuga de informação considerados.

Palavras-chave: JavaScript Analysis, Information Flow
Abstract

JavaScript is one of the most relevant languages for client side applications as can be seen by the number of repositories in github (http://github.info). Once limited to form validation and other small tasks, today it has evolved into a major component of the Web. JavaScript is a very dynamic language, as it allows the creation of both variables and functions in runtime. However this dynamic nature is a double-edged sword as it makes it's analysis a complex task. Current tools for JavaScript analysis are limited, mostly due to the combination of the aforementioned dynamism (forcing tools to analyse small subsets of the language) and the heterogeneity of browsers (which interpret JavaScript differently). This lack of analysis tools makes the development of secure JavaScript code a non-trivial challenge.

In this thesis, we propose a solution that is based on information flow techniques, which provide strong guarantees while being independent of the browser. We present a tool that consists of an information flow monitor, that aims to keep confidentiality by catching both explicit and implicit information leaks. This monitor analyses the intermediate language JSIL, a simple goto language created for JavaScript Verification. In order for a program to be analysed it needs to be compiled into JSIL through the JS-2-JSIL compiler, which was enhanced in order to contain commands that support monitors. These new commands allows us to label variables with security levels and to manipulate these labels. Finally, we take advantage of the fact that JSIL supports symbolic execution to analyse the program through traversing multiple execution paths.

We propose a monitor that aims to keep confidentiality by catching implicit information flow leaks. To do so we formally define the syntax and the semantics of the rules the monitor has to follow, as well as implement them. Finally we run the execution through a test suite of unitary tests to evaluate the monitor, proving we can catch all types of information flow leaks under consideration in JSIL.

Keywords: JavaScript Analysis, Information Flow
# Contents

Acknowledgments ......................................................... v
Resumo ........................................................................ vii
Abstract ..................................................................... ix
List of Tables ................................................................. xiii
List of Figures ................................................................. xv
Nomenclature ................................................................. 1
Glossary ........................................................................ 1

1 Introduction ................................................................. 1
1.1 Motivation ................................................................. 1
1.2 Running Example ..................................................... 2
1.3 Tool Chain ............................................................... 4
1.4 Objectives ................................................................. 4
1.5 Contribution ............................................................. 6
1.6 Thesis Outline .......................................................... 6

2 Related Work and Background ........................................ 7
2.1 Information Flow Monitors ......................................... 7
   2.1.1 Basics Concepts of Information Flow .................. 7
   2.1.2 Information Flow Analyses ............................... 8
   2.1.3 Information Flow Monitors ............................... 9
2.2 JavaScript Security .................................................. 10
   2.2.1 Overview of JavaScript .................................... 10
   2.2.2 JavaScript Analysis ......................................... 11
   2.2.3 JavaScript Information Flow Security ................. 12
2.3 Symbolic Execution .................................................. 12
   2.3.1 Basics .......................................................... 12
   2.3.2 Symbolic Execution for Javascript .................... 13

3 JSIL ........................................................................... 15
3.1 Syntax .................................................................... 15
3.2 Semantics ............................................................... 16
List of Tables

3.1 Semantic Domains .................................................. 17

4.1 Low Projection for JSIL Semantic Domains \((h, m, \rho) \upharpoonright_{(sh, sm, sp)} \) \(\triangleq (h \upharpoonright_{sh}, m \upharpoonright_{sm}, \rho \upharpoonright_{sp})\) ........ 28

4.2 Naive Approach vs No Sensitive Upgrade .......................... 32

6.1 JSIL test suite ....................................................... 66

6.2 JavaScript test suite ................................................... 68

A.1 JSIL simbolic Semantics .............................................. 77
# List of Figures

1.1 Example of Scope in a Prototype ................................................. 2  
1.2 Example of Scope in JS .......................................................... 2  
1.3 Example Program ................................................................. 3  
1.4 Code Snippet `getRightBit1` .................................................... 4  
1.5 Code Snippet `getRightBit2` .................................................... 4  
1.6 Code Snippet `getBits` ........................................................... 5  
1.7 pre-processing phases of the program ........................................ 5  
1.8 Architecture of the monitor ..................................................... 5  
2.1 Symbolic Execution example .................................................... 13  
3.1 Semantic of Basic Commands: \( \{h, m, \rho, bc\} \xrightarrow{\alpha} \{h', m', \rho'\} \) .................................... 17  
3.2 Execution for Control Flow Commands: \( \{h, m, \rho, cs, i\} \xrightarrow{\alpha} \{h', m', \rho', cs', j\} \) .................. 19  
3.3 Semantics of Security Commands: \( \{\rho, sc\} \Downarrow \) .................. 20  
3.4 Monitored Semantics ............................................................. 21  
3.5 Event Labels ........................................................................... 21  
3.6 Running Example in JSIL .......................................................... 25  
3.7 Running example when x is odd ............................................... 23  
3.8 Running example when x is even ............................................. 24  
4.1 Example JSIL heap .................................................................... 30  
4.2 Example monitored JSIL security heap ...................................... 30  
4.3 Low projection for a low level observer ..................................... 31  
4.4 Monitored Semantics Upgrades: \( \{sh, sm, sp, scs, pc\} \xrightarrow{\alpha_{mon}} \{sh', sm', sp', scs', pc'\} \) ................. 34  
4.5 Monitored Semantics: \( \{sh, sm, sp, scs, pc\} \xrightarrow{\alpha_{mon}} \{sh', sm', sp', scs', pc'\} \) .................. 36  
4.6 Example .................................................................................. 41  
4.7 Example of running a program - instrumentalized ....................... 42  
4.8 Example of running a program - Monitor ................................... 43  
4.9 Example of running a program - Monitor ................................... 44  
4.10 Running Example in JSIL ........................................................ 45  
4.11 Running Example with upgrades JSIL ..................................... 46  
4.12 Running example when x is odd ............................................. 47
Chapter 1

Introduction

JavaScript is one of the most used language for client-side web applications. Indeed it is used by 94.8% of websites \(^1\) and is the most active language in both GitHub \(^2\) and StackOverflow \(^3\), although initially it was mainly used to validate forms and other small tasks.

Since each browser has different mechanisms for handling security, it becomes very hard to develop a secure application that can work in all browsers. This problem can be solved by using techniques that work independently of the browser. A natural approach to handle this would be by using a verification tool based on enforcing information flow security, which consist in analysing the way information moves through a program.

1.1 Motivation

Information is a critical asset in most businesses, hence nowadays software security and protecting data is of utmost relevance. Minimizing the vulnerabilities directly corresponds to a reduction on the impact that a possible attack could have in normal day-to-day operations. Companies want the insurance that a specific interaction with some software is secure and doesn’t represent any damage to themselves.

In security management Risk is the combination of the probability of an event and its consequence. In general, this can be explained\(^4\) as a function of the likelihood of an attack times the impact such an attack would have. "An earthquake isn’t able to figure out how to topple structures constructed under some new and safer building code" \(^5\) the fact that there is such a ever-adapting reality requires the solutions to be as compatible and encompassing as possible so that organizations can use the same tool for different situations. Here we aim to provide a tool that will be integrated into the software development cycle aiming to get strong security properties that can be demonstrated.

From a risk based approach to application security, the popularity and widespread of JavaScript makes it of the utmost importance to develop techniques able to analyse it. JavaScript is considered

---

\(^1\) w3techs.com/technologies/details/cp-javascript/all/all
\(^2\) http://github.info
\(^3\) https://exploratory.io/viz/Hidetaka-Ko/94368d12800a?cb=1469037012628
\(^4\) owasp.org/index.php/OWASP_Risk_Rating_Methodology
\(^5\) https://www.schneier.com/essays/archives/2013/08/our_decreasing_toler.html
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
}
Person.prototype.nationality = "English";

Figure 1.1: Example of Scope in a Prototype

// code here can NOT use carName
function myFunction() {
    var carName = "Volvo";
    // code here CAN use carName
}

Figure 1.2: Example of Scope in JS

to be a Dynamic language as it can create both variables and functions in runtime. Due to this dynamic nature and its complex semantics the development of secure code is particularly difficult. Whole language analysis is still an open problem.

In order to deal with these characteristics we use information flow techniques, which are based in analysing the way information flows between different variables during code execution. These techniques usually aim to prevent leaks, this means private information flowing into public information. Information flow techniques are specially useful to enforce both the confidentiality and the integrity of data, as they focus on the flow of the information they are applicable at the application level allowing for an abstraction of the browser itself.

JavaScript is a very complex language, and every new version increases its complexity. This complexity can bring strange interaction that can leak information. In order to analyse JavaScript we have opted to use the intermediate language JSIL which simulates the behaviour of the JavaScript language. Although the compilation to JSIL increases the size of the program, this intermediate Language has much simpler semantics than the JavaScript language simplifying the future the analysis. At this point it would be possible to use symbolic execution, which allow to analyse the whole program.

1.2 Running Example

In order to ease the comprehension of this section, we need to introduce some basic concepts of JavaScript programs. JavaScript is a high level object oriented language. All JavaScript objects inherit properties and methods from a prototype.

For instance a Person object inherits from Person.prototype. The prototype property allows the addition of new properties and objects to constructors as can be seen in figure 1.1.

Scope determines the accessibility (visibility) of variables. In JavaScript there are two types of scope. The Local scope containing the variables declared inside a function; and the Global scope, containing the variables declared outside a function, as can be seen in the Figure 1.2.

In order to illustrate the importance of information flow analysis for JavaScript programs, let us con-
Consider the code snippet in Figure 1.3, which was obfuscated. Upon careful examination, we determine that the program copies the value of its input in a way that is not detected by standard static analyses, and hard to detect manually.

Figures 1.4-1.6 show the original version of the program given in Figure 1.3. It consists of 5 functions: (1) `getRightBit1` and (2) `getRightBit2` that return the last bit of their respective inputs; (3) `getBits`, that returns an array containing the bits of its input; (4) `bitsToNumber`, that converts the array of bits given as input into a number; and (5) `getRandomBit`, that returns a random boolean.

Unlike functions `getBits`, `bitsToNumber`, and `getRandomBit`, whose behaviour is apparent from the code, functions `getRightBit1` and `getRightBit2` do not achieve their (common) goal in a clear way. Instead, they take advantage of obscure corner cases of the JS semantics, which are ignored by most programmers and static analyses. Below, we give a detailed explanation of how functions `getRightBit1` and `getRightBit2` achieve their goals.

Starting by `getRightBit1`, the function creates an array `a` with 3 elements ["a", "b", "c"]. Then, if the input is even, the function updates the property "1" of `a` to the value "d", also setting it to be non-configurable. In a nutshell, non-configurable properties cannot be deleted. Afterwards, the function `tries` to set the length of `a` to 1, meaning that all the entries but the first must be deleted. However, if the function changed the property "1" of `a` to be non-configurable, then "1" cannot be deleted, resulting in a final length of 2 instead of 1. Finally, the function returns 1 if the length of `a` is equal to 1, or 0 otherwise, effectively leaking the last bit of its input.

Function `getRightBit2` starts by defining two variables `op` and `f`. Variable `op` holds an empty object. Variable `f` is initialised as a function. In JavaScript, every function can be used as a constructor. In fact, by assigning `op` to `f.prototype`, every object created using `f` as a constructor will have `op` as its prototype. Function `getRightBit2` then creates a new object `o` using `f` as its constructor. Now, if the input is even, `getRightBit2` creates a new property "foo" in `op` with value 0, also setting it to be non-writable. In a nutshell, non-writable properties cannot be overwritten. Then, `getRightBit2` tries create a new property "foo" in `o` with value 42. If `x` is odd (meaning that "foo" does not exist in the prototype), `getRightBit2` is able to execute the property assignment successfully and returns 1. However, if `x` is even, the property "foo" does exist in the prototype; furthermore, it is set to be non-writable. In this case, the property assignment is not allowed and the JS engine throws an error, causing `getRightBit2` to return 0.

The function `getBits` iterates over the bits of its input. For each iteration, it calls either `getRightBit1` or `getRightBit2` depending on the output of `getRandomBit`. The function then returns an array containing all the bits of its input, which can easily be transformed into a number through `bitsToNumber`.

By analysing this example, one can easily see that, by taking advantage of subtle details of the
function getRightBit1 (x) {
    var a = [ 'a', 'b', 'c' ];
    if (x % 2 === 0) {
        Object.defineProperty (a, '1', { value: d, configurable: false } );
    }
    a.length = 1;
    return (a.length === 1) ? 1 : 0
}

Figure 1.4: Code Snippet getRightBit1

function getRightBit2 (x) {
    var op = {};
    var f = function () { }; 
    f.prototype = op;
    var o = new f ();
    if (x % 2 === 0) {
        Object.defineProperty (op, 'foo', { value: 0, writable: false })
    }
    try {
        o.foo = 42;
        return 1
    } catch (e) {
        return 0
    }
}

Figure 1.5: Code Snippet getRightBit2

JavaScript semantics, it is trivial to copy secret values to insecure variable in hazy ways. Information flow analysis enables the detection and prevention of such type of exploits.

1.3 Tool Chain

We propose to analyse JavaScript code by first compiling it to JSIL, a simple static language that encapsulates the dynamic interactions of JavaScript, through the JS-2-JSIL [1] tool. This new JSIL code will have to be interpreted into Racket, this interpretation can be done through the JSIL-2-RKT [2] tool. We use the Racket language in order to make use of the symbolic execution engine Rosette which was built upon this language. This sequence is illustrated in 1.7. JSIL-2-RKT goes through the JSIL code to create a Racket code that can easily be interpreted. Now that the code has been compiled into Racket, we apply our monitor to analyse the information flow of the program.

The architecture of the information flow monitor can be seen in Figure 1.8. The monitor receives the Racket program and its input. The monitor is coupled with the Racket interpreter in order to enforce the flow of information within the program.

1.4 Objectives

In this thesis we aim to develop a JavaScript analysis monitor that uses both information flow techniques and symbolic execution techniques. These information flow techniques allow to abstract the JavaScript analysis from the application level allowing this monitor to be used in any browser. We aim to create a
function getRandomBit () {
    var a = Math.random >= 0.5;
    return a
}
function getBits (x) {
    var bits = [];
    while (x > 0){
        var aux = getRandomBit ();
        if (aux) {
            var rbit = getRightBit1 (x);
        } else {
            var rbit = getRightBit2 (x);
        }
        bits.push(rbit);
        x = x >> 1;
    }
    return (bits.length === 0) ? [ 0 ] : bits.reverse();
}
function bitsToNumber (bits) {
    var ret = 0;
    for (var i = 0; i < bits.length; i++) {
        ret = 2*ret + bits[i]
    }
    return ret
}

Figure 1.6: Code Snippet getBits

![Diagram showing the pre-processing phases of the program](image)

Figure 1.7: pre-processing phases of the program

![Diagram showing the Architecture of the monitor](image)

Figure 1.8: Architecture of the monitor
monitor that catches obscure information flow leaks that other monitors are unable to catch.

In the long run, we plan to apply the developed mechanism to the TAGUS voting system that is in development for Instituto Superior Técnico.

1.5 Contribution

The main contribution of this thesis is an information flow monitor that catches implicit information flow leaks in JavaScript. To do so we have compiled the JS program into JSIL, which is a simple goto language that is much easier to analyse than JS. Therefore our contribution consists of:

- Extending the JSIL language to support a monitor.
- Formal definition of the Syntax and Semantics of our information flow monitor, which are thoroughly explained in chapter 4.
- Development of the monitor, which we implemented in Racket. This is thoroughly explained in chapter 5.
- Validation of the correctness and robustness of the monitor, by running our monitor through a battery of tests designed encode different forms of implicit information flows. This process is explained in chapter 6.

1.6 Thesis Outline

We start by an overview of the state of the art of JavaScript analysis, information flow and symbolic execution (Chapter 2). We then present the intermediate language JSIL (Chapter 3), which is an intermediate representation that captures the particular features of JavaScript, and propose a general monitored semantics. We then define a JSIL monitor using the semantics proposed (Chapter 4). We describe the implementation of the monitor (Chapter 5), comparing the semantic rules to the code that we have developed. We evaluate our monitor (Chapter 6) by running tests from a battery we created. Finally, we review the various stages for the application of our tool, discuss future work, and conclude (Chapter 7).
Chapter 2

Related Work and Background

In this chapter we will look over the basics and the state of the art of the major topics of this thesis. We will start by analysing information flow techniques and their basic concepts, and refer to studies of different kinds of information flow monitors. We then provide an overview of the major characteristics of JavaScript, and discuss existing works on its analysis as well as on enforcement of information flow security. Finally we give a brief overview of symbolic execution basic concepts, and summarise works on JavaScript.

2.1 Information Flow Monitors

In this section we will first start by introducing a few basic concepts of information flow. Then we will study different approaches to information flow analyses. Finally we will analyse information flow monitors that have already been built.

2.1.1 Basics Concepts of Information Flow

The starting point in secure information flow analysis is the classification of variables into different security levels in order to maintain a security property such as confidentiality or integrity. The most basic distinction is to classify some variables as L (low security, public information); and other variables as H (high security, private information). The security goal is to prevent information in H variables from being leaked improperly [3], i.e. to influence the information that is available in L variables.

Considering that the level L is contained within H (L < H), then we would allow flows from L to L, from H to H, and from L to H, but we would disallow flows from H to L. Meaning that an insecure flow of information, or interference, can be said to occur when the initial values of high variables influence the final value of low variables.

The following example is an explicit illegal flow, as a public variable (whose level is L) was assigned the value of a secret variable (whose level is H):

\[
\text{public} = \text{secret};
\]

(2.1)
While this example should be legal, as there is no leak of secret information:

\[ \text{secret} = \text{public}; \]  

(2.2)

The following example represents an implicit flow [4], this means that no condition on a private variable (secret) will change a public variable, this would give an attacker who could read the variable public the ability to make assumptions about the secret variable.

\[ \text{if}(\text{secret} \mod 2 == 0) \text{public} = 1; \text{public} = 0; \]

(2.3)

We may be interested in protecting integrity rather than confidentiality. If we view some variables as containing possibly tainted information, then we may wish to prevent information from such variables from flowing into untainted, in order to prevent the integrity of such variables.

the following example is an explicit illegal flow:

\[ \text{untainted} = \text{tainted}; \]

(2.4)

While this example should be legal:

\[ \text{tainted} = \text{untainted}; \]

(2.5)

When a variable or field is assigned a value of different sensitivity than the one it currently contains, it is necessary to apply the maximum of the two security labels to the stored value. However increasing the security level of the labels can create the phenomenon known as label creep, which is a problem for dynamic enforcement mechanisms. Label creep can make dynamic labelling systems too restrictive for dynamic systems [5].

### 2.1.2 Information Flow Analyses

The challenge of enforcing secure information flow is due to prevent the execution of programs that can create illegal dependencies between the resources on which they operate. In order to overcome this there are multiple approaches:

- **Static**: The major advantages of this approach consist of an ability to track implicit flows accurately, as well as being able to precisely control information flow with little runtime overhead [6]. Some of the implementations use theorem provers such as [7], and type checking techniques [4].

- **Dynamic**: Purely Dynamic techniques, such as [8], suffer from multiple disadvantages as not only they suffer from a runtime overhead due to the fact they are run at runtime, but they are also less precise. This is due to purely dynamic approaches only having access to a single program execution, while its static and hybrid counterparts use static techniques to analyse multiple. They are however very intuitive to apply to dynamic languages.
• **Hybrid**: Due to the dynamic nature of JavaScript it is very hard to design a purely static analysis. This kind of monitor uses static techniques to reduce the issues of dynamic techniques, such as [9] that uses program analysis combined with runtime checks to verify widgets. This type of monitor has the big advantage of reducing the overhead that runtime monitors impose. This kind of monitor must either statically or dynamically estimate the resources that are being created in the program paths that were not taken. Hybrid information flow monitors such as [10] [11], use static analysis to reason about the implicit flows. Additionally this kind of monitor has been found out to be more permissive than both purely static and purely dynamic counterparts [12].

### 2.1.3 Information Flow Monitors

Flow-sensitive monitors for enforcing noninterference can be divided into two categories:

• **purely dynamic** [13], which impose a runtime overhead [14] as these strategies do not rely on any kind of static analysis, such as [15] [16], [17], and [18]. Additionally purely dynamic monitors are too conservative.

• **hybrid monitors**, mixing runtime monitoring with static analysis, such as [19], [10], and [11].

The authors of [16], [17], and [18] propose alternative strategies for designing sound purely dynamic information flow monitors.

• **no sensitive upgrade**: [15, 16], that forbids the update of public resources inside private contexts.

• **permissive upgrade**: [17] strategy, that allows sensitive upgrades to take place, but marks the resources upgraded in sensitive contexts and forbids the program to branch depending on the content of these resources.

• **multiple facet**: this strategy surpasses the limitations of the first two by the use of multiple faceted values. The main concept of this strategy is that values appear differently at different security levels.

• **multiple execution**: [20] [21] [22], The main idea is to execute a program multiple times, once for each security level, using special rules for I/O operations. Where outputs are only produced in the execution corresponding to their level and inputs are only used in executions corresponding to their security level or higher.

Whilst dynamic monitors have access to the aforementioned strategies, hybrid monitors must estimate the set of resources that are created in untaken program branches.

**Coarse Grained Information Flow Monitors.** In the past years, coarse-grained information flow moni-
tors [23, 24, 25, 26] have emerged as an alternative to fine-grained information flow monitors. The main advantage of this type of monitoring is the ease of use in integrating them with existing languages [23, 25].
Coarse-grained monitors are designed so that the level of the program counter is represented by an upper bound on the levels of all data observed or modified. Raising the current level of the program counter allows computations to read data in a very flexible way at the cost of creating the phenomenon known as label creep [27], where the security labels of a monitor are raised to a point where the program is restricted to write only to high confidentiality levels, to an unrealistic degree. Additionally coarse grained monitors are less precise, although recently this cost has been reduced [28].

Monitoring Secure Information Flow in the Browser. The first information flow monitor to implement for a realistic core of JavaScript [13] introduced the notions of existence level and structure security level for the labeling of JavaScript objects. Since the monitor [13] is purely dynamic, it suffers from the limitations of being very conservative [14]. To overcome these limitations, [29] show how to use tests in order to boost the permissiveness of the monitor.

Despite targeting JavaScript, the monitors [13] [29], do not model the reactive aspect of client-side web applications. [30] presented a definition of reactive noninterference, as well as a monitor for enforcing it. Later, [31] proposed an enforcement mechanism for reactive noninterference based on secure multi-execution [32].

2.2 JavaScript Security

We first start by giving an overview of JavaScript's properties, then we see what makes analysing it such a challenge and how it has been done. Finally we study the information flow techniques used in JavaScript.

2.2.1 Overview of JavaScript

JavaScript is an object-based language. However, instead of having classes, every non-native object has a prototype, which is also an object, from where it can inherit properties. Therefore, prototypical inheritance is a form of delegation, in the sense that an object dispatches to its prototype the requests that it does not know how to handle.

Objects are the central datatype of JavaScript. But, unlike other to class-based languages where the fields of an object are restricted by the class to which it belongs , a JavaScript object is an unrestricted mapping from strings to values. The strings in the domain of an object are called its properties. In JavaScript there are two types of objects: those that are defined by the programmer and those that are provided by the language runtime (internal objects). Another important feature of JavaScript is that programs are allowed both to dynamically add new properties to the domain of an object and to delete existing ones.

In order to look-up the value of a property "x" of an object bound to a variable o, the JavaScript engine first checks whether "x" belongs to the properties of the object bound to o. If "x" is in the object bound to o, the property look-up yields the value with which that object associates property "x". Otherwise the
engine delves into the prototype in search of an object that defines a property named "x".

The sequence of objects that can be accessed from searching the prototypes of an object is called a prototype-chain. This property brings a new kind of possible implicit leaks, when combined with the dynamic nature of javascript creates a new set of challenges for information flow control. Some of these challenges [33] are:

- **Extensible Objects**: in Core JavaScript, the programmer can dynamically add and remove properties from objects.

- **Leaks through Prototype mutations**: The fact that a prototype of an object is allowed to change at runtime may be exploited to encode security leaks.

- **Leaks though the existence of properties**: as in JavaScript, a program can dynamically add and remove properties from objects and a check whether a property is defined in the prototype-chain. Meaning that the mere existence of a property in the domain of an object may disclose confidential information.

### 2.2.2 JavaScript Analysis

Hedin et al [13] have been the first to propose an information flow monitor for a realistic core of JavaScript, introducing the notion of existence levels to deal with the leaks through the checking of the existence of properties. Their approach is dynamic, even though this kind of approach is well suited to analyse dynamic languages they bring a big runtime overhead [14].

Contrary to this approach, the static perspective avoid the complex semantics of JavaScript by focusing on isolation properties [34], which are easier to enforce than noninterference [35]. The issue of having to deal with names being able to be computed at runtime has already been dealt with in different ways [34] [8].

In order to deal with the runtime problems associated with the dynamic approach, one could use static techniques creating an hybrid approach [36]. This kind of analysis avoids rejecting programs due to imprecise typing. In [37] an hybrid typing analysis for enforcing secure information flow in a core of JavaScript was presented.

Given the dimension of the programming languages used in the real-world, the first approaches to analyse these languages usually focus on a subset of the core language, focusing on the most important characteristics. An alternative approach is to use an intermediate representation try to encapsulate the most relevant properties of JavaScript, with the goal of reducing the difficulty of the analysis.

Intermediate representation analysis can be broadly divided into two categories. The first one is based on syntax-directed analyses, such as [38], which are usually best suited for high level analysis. The other approach focus on analysing the control flow graph of a program [39], usually the target of symbolic execution.
2.2.3 JavaScript Information Flow Security

The first information flow monitor implemented for a realistic core of JavaScript [13] introduced the notions of existence level and structure security level for labelling objects. The main mechanisms for securing information flow in JavaScript [40] [13] [41] are mostly-dynamic due to the dynamic nature of the language. Usually these kinds of tool consider only a subset of the language, such as [1] [42]. The tool presented in [43] allows the analysis of information flows for the whole JavaScript language. In comparison the monitor we present in this thesis allows it to be executed symbolically, effectively allowing it to explore more execution paths.

In order to deal with the problem of dynamically adding or removing property names at runtime there are multiple solutions, such as not considering in the subset certain lookups in the subset of the language we want to analyse [34].

2.3 Symbolic Execution

Many security and software testing applications require checking certain properties of a program hold for any possible usage scenario. A straightforward and popular approach would be to test the program using different inputs. Not only is this approach time consuming but it tests path of execution that would be unfeasible. Symbolic execution provides an elegant solution to the problem, by systematically exploring “all” possible execution paths at the same time by means of symbolic inputs.

2.3.1 Basics

Symbolic execution consists of using symbolic values instead of concrete values to analyse to which possible paths a program takes. Symbolic execution tools can be divided into two main groups:

- **Static**: such as [44] [45] [46] [47], explore the entire symbolic execution tree up to an established bound, providing bounded verification guarantees.

- **Dynamic**: such as [48] [49] [50], are aimed at automatic test generation and generally do not provide any verification guarantees. These tools normally work by pairing up a concrete execution with a symbolic execution, creating an hybrid approach known as concolic execution, in order to allow the symbolic execution to fall back to the concrete execution whenever it produces symbolic formulae that are not supported by the constraint solver.

This approach brings its own set of challenges, such as path explosion. A symbolic executor can fork at every branch of the program, which can lead to an exponential amount of states. This will impact both the time and space requirements, therefore techniques that can reduce these effects are needed. For instance [49] was the first dynamic SE tool with support for summaries. It tests functions in isolation in a bottom-up manner, encodes test results as first-order constraints and re-uses these constraints as summaries in the testing of other functions. This approach is particularly useful when traversing the code multiple times.
In order to illustrate the importance of symbolic techniques we borrow the example\(^1\) in the Figure 2.1. Upon initializing the \(x\) and \(y\) variables there is an if forking the execution into two, the first where \(X\) is bigger than \(Y\) and the second where it is not. Note that the path condition is updated in both to have the information associated with the value of \(X\) and \(Y\). Now we will analyse the case where it is true, as both cases will be very similar. From line 3 to 5 there are a sequence of assigns that result in having the original value of \(x\) assigned to \(y\) and the original value of \(y\) assigned to \(x\). Now the second if will function like the first one, but notice that in the case it evaluates to be true \(X\) would need to be both larger than and lower than \(Y\), making this state impossible. Meaning that the second if could only be false.

### 2.3.2 Symbolic Execution for Javascript

The dynamic nature of JavaScript and its complex semantics make it a difficult target for logic-based verification. The majority of the existing bug-finding symbolic execution tools for JavaScript are whole-program and target specific bug patterns, such as [51] [52] [53]. These tools are fully automatic and aim at code in the large, primarily focusing on scalability and coverage issues.

Meanwhile, [54] developed a general-purpose symbolic execution tool for JavaScript, which does not follow the semantics of the language precisely. In contrast [55] have developed a framework, which was applied to whole-program symbolic testing of real-world JavaScript libraries, that assists in code testing by providing counter-models. Later [5] developed a verification toolchain built upon it.

\(^1\) Zvonimir Rakamarić. University of Utah, "Lecture 6: Symbolic Execution"
Chapter 3

JSIL

JavaScript is hard to analyse due to its dynamic nature and complex semantics. Therefore, instead of analysing JS programs directly, we first compile them to JSIL [1] and analyse the obtained JSIL program. Although the JSIL program has more lines than the original JavaScript code, as JSIL has a simpler semantics it facilitates the analysis. JSIL is a simple goto language with top-level procedures and commands operating on object heaps. In this chapter, we first explain the syntax and the semantics of the intermediate JSIL language (Section 3.1 and Section 3.2). Then, we introduce a novel general monitored semantics for JSIL, which is parametric, on an arbitrary monitor given as input (Section 3.3). Finally, we show the JSIL execution of a stylized compilation of the running example (Section 3.4).

3.1 Syntax

JSIL expressions include JSIL literals, JSIL variables x, and a variety of unary and binary operators. The unary and binary operators contain:

- the standard arithmetic (+, -, *, /, %);
- comparison (=, <, (lexicographic comparison of strings));
- boolean (not, and, or);
- list operators [head, tail, length, :: (cons), @ (list append), nth (n-th element of a list), ord (returns true when applied to a list of lexicographically ordered strings)) and string (length s, @ s (string concatenation), nth (n-th element of a string));
- operators that convert string to numbers and the other way around (toString and toNumber), additionally there is the typeOf operator that returns the type of an expression.
Syntax of the JSIL Language

Numbers: \( n \in \mathbb{N} \)  
Booleans: \( b \in \mathcal{B} \)  
Strings: \( s \in \mathcal{S} \)  
Locs: \( l \in \mathcal{L} \)  
Vars: \( x \in \mathcal{X} \)

Types: \( \tau \in \mathcal{T} \)  
Values: \( v \in \mathcal{V}_{\text{JSIL}} := n \mid b \mid s \mid \text{undefined} \mid \text{null} \mid l \mid \tau \mid \text{fid} \mid \text{empty} \)

Expressions: \( e \in \mathcal{E} \)  
JSIL ::= \( n \mid b \mid s \mid \text{undefined} \mid \text{null} \mid l \mid \tau \mid \text{fid} \mid \text{empty} \)

Basic Commands: \( bc \in \mathcal{B}_{\text{cmd}} := \text{skip} \mid x := e \mid x := \text{new}() \mid x := e; e \mid e := e \mid e := e \mid x := \text{hasField}(e,e) \mid x := \text{getFields}(e) \mid \text{assume}(e) \mid \text{assert}(e) \)

Security Commands: \( sc \in \mathcal{S}_{\text{cmd}} := \text{upgVar}(x,\sigma) \mid \text{upgProp}(e_1,e_2,\sigma) \mid \text{upgProp}(e,e,\sigma) \mid \text{upgStruct}(e,\sigma) \mid \text{upgMtd}(e,\sigma) \mid \text{merge}(\bar{i}) \)

Commands: \( c \in \mathcal{C}_{\text{cmd}} := \text{sc} \mid \text{goto} i \mid \text{goto}[e] i, j \mid x := e(\bar{e}) \)

Procedures: \( \text{proc} \in \mathcal{P} := \text{proc} \text{fid}(x)\{c\} \)

Program : \( \text{fid} \rightarrow \text{proc} \)

JSIL basic commands enable the interaction with a single variable store and the manipulation of extensible objects. Basic commands and do not affect the control flow. They include: \text{skip}, variable assignment, object creation, property assignment, property deletion, has field, property lookup, metadata, and two special commands, assume and assert, essential for symbolic execution, but with trivial concrete semantics.

JSIL security commands, enable the manipulation of the security levels associated with the various types of resources manipulated by the language, them being: object properties, object metadata and program variables. This kind of command does not affect the control flow.

JSIL control flow commands include the basic commands, the security commands and several commands related to control flow: conditional gotos, unconditional gotos, dynamic procedure calls.

A JSIL program \( p \in \mathcal{P} \) can be seen as a set of top-level procedures of the form \( \text{proc} \text{fid}(\bar{e})\{c\} \), where \( \text{fid} \) is the procedure name, \( \bar{e} \) are its formal parameters, and its body \( \bar{e} \) is a command list consisting of a sequence of JSIL commands. Every JSIL program contains a special procedure \text{main}, denoting the entry point of the program. JSIL procedures return via two dedicated indexes, \( i_{\text{nm}} \) and \( i_{\text{er}} \), using two dedicated variables, \( \text{ret} \) and \( \text{err} \). If a procedure reaches the \( i_{\text{nm}} \) index, it returns normally with the return value denoted by \( \text{ret} \); when it reaches \( i_{\text{er}} \), it returns an error, with the error value denoted by \( \text{err} \).

### 3.2 Semantics

Table 3.1 introduces the semantic domains of JSIL. A JSIL heap, \( h \in \mathcal{H}_{\text{ap}} \), maps a pair consisting of location \( l \in \mathcal{L}_{\text{oc}} \) and a property \( p \in \mathcal{P}_{\text{rops}} \) to a value \( v \in \mathcal{V}_{\text{al}} \). A JSIL metadata, \( m \in \mathcal{M}_{\text{etadata}} \), maps a location \( l \in \mathcal{L}_{\text{oc}} \) to a value \( v \in \mathcal{V}_{\text{al}} \). A JSIL store, \( \rho \in \mathcal{S}_{\text{tore}} \), maps variables \( x \in \mathcal{X} \) to values \( v \in \mathcal{V}_{\text{al}} \). A JSIL call stack, \( cs \in \mathcal{CS} \), is either empty or a list of tuples of the form \( (f,\rho,x,i,j) \), where: \( f \) is a procedure call; \( \rho \) is the store; \( x \) is the variable to which the result of \( f \) is to be assigned; \( i \) is the index of the command the program should jump to if \( f \) returns normally while \( j \) is the index of the command the program should jump to if \( f \) returns an error. A semantic configuration \( \Omega \) consists...
The semantics of JSIL basic commands is given in Figure 3.1 in a small-step style; semantic transitions have the form \( \{h, m, \rho, bc\} \xrightarrow{\omega} \{h', m', \rho'\} \), where:

- \( \{h, m, \rho, bc\} \) is the initial configuration consisting of a heap, a metadata table, a store and the basic command to execute;
- \( \omega \) is an event label to be given to the information flow monitor, which will be explained in the next section;
- \( \{h', m', \rho'\} \) is the final configuration consisting of the final heap, metadata and store.

Below we give a explanation of the semantics of each command.
Skip: When evaluating the command `skip`, the semantics does nothing.

Object Creation: When evaluating the command `x := new(e)`, the semantics first evaluates the expression `e`, obtaining a value `v`. Then, it generates a fresh location `l` and associates it with `v` in the metadata table, `m`. Finally, it updates the value of `x` to the newly generated location `l`.

Assign: When evaluating the command `x := e`, the semantics first evaluates the expression `e`, obtaining a value `v`. Then it updates the current store, setting `x` to `v`.

Lookup: When evaluating the command `x := [e1, e2]`, the semantics first evaluates the expressions `e1` and `e2`, obtaining a location `l` and property `p`, respectively. The semantics then search the heap for the value `v` associated with the entry `(l, p) (h(l, p) = v)`. Finally, the value of `x` will be updated to `v`.

Property Assign 1: When evaluating the command `[e1, e2] := e3`, the semantics first evaluates the expressions `e1`, `e2` and `e3` to a location `l`, a property `p` and a value `v`, respectively. Then the semantics finds in the heap the entry associated with `(l, p)` updating it to `v`.

Property Assign 2: When evaluating the command `[e1, e2] := e3`, the semantics first evaluates the expressions `e1`, `e2` and `e3` to a location `l`, a property `p` and a value `v`, respectively. Then the semantics does not find in the heap an entry for `(l, p)`, updating the heap with a new entry where `(l, p)` is assigned the value `v`.

Delete: When evaluating the command `delete(e1, e2)`, the semantics first evaluates the expressions `e1` and `e2`, obtaining a location `l` and a property `p` respectively. Then the semantics finds in the heap the entry for `(l, p) (h(l, p) = v)`, and deletes it.

Has Field - True: When evaluating the command `x := hasField(e1, e2)`, the semantics first evaluates the expressions `e1` and `e2`, to a location `l` and a property `p` respectively. The Semantics finds the entry in the heap for `(l, p) (h(l, p) = v)`, updating `x` to true.

Has Field - false: When evaluating the command `x := hasField(e1, e2)`, the semantics first evaluates the expressions `e1` and `e2`, to a location `l` and a property `p` respectively. The Semantics does not find the entry in the heap for `(l, p)`, updating `x` to false.

Get Fields: When evaluating the command `x := getFields(e)`, the semantics first evaluates the expression `e` to a location `l`. The Semantics then searches the heap for all properties of `l` `h(l[p]) = v[i=1]`, updating the value of `x` to `p[1][n]`.

Metadata: When evaluating the command `x := metadata(e)`, the semantics first evaluates the expressions `e` to a location `l`. The Semantics searches the metadata for the value `v` assigned to `m(l) = v)`. Finally, `x` is updated to `v`.

### 3.2.2 Control Flow Commands

The semantics of JSIL Control flow commands is given in Figure 3.2 in a small-step style; semantic transitions have the form `{h, m, p, cs, i} \xrightarrow{\omega} {h', m', p', cs', j}`, where:

- `{h, m, p, cs, i}` is the initial configuration consisting of a heap, a metadata table, a store, a call stack and the current index of the program;
- $o$ is an event label to be given to the information flow monitor, which will be explained in the next section;
- $\{h', m', \rho', cs', j\}$ is the final configuration consisting of the final heap, metadata, store, call stack and the index of the next command.

<table>
<thead>
<tr>
<th><strong>Basic Command</strong></th>
<th><strong>Goto</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{cmd}(i) = be \in Bcmd$</td>
<td>$\text{cmd}(i) = \text{goto } j$</td>
</tr>
<tr>
<td>${h, m, \rho, \text{cs}, i} \xrightarrow{\text{BeCmd}} {h', m', \rho'}$</td>
<td>${h, m, \rho, \text{cs}, i} \xrightarrow{\text{goto } j} {h', m', \rho', \text{cs}, i + 1}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Cond. Goto - True</strong></th>
<th><strong>Cond. Goto - False</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{cmd}(i) = \text{goto } [e] j$, $k$ $[e]_{o} = \text{true}$</td>
<td>$\text{cmd}(i) = \text{goto } [e] j$, $k$ $[e]_{o} = \text{false}$</td>
</tr>
<tr>
<td>${h, m, \rho, \text{cs}, i} \xrightarrow{\text{Cond. Goto - True}} {h, m, \rho, \text{cs}, k}$</td>
<td>${h, m, \rho, \text{cs}, i} \xrightarrow{\text{Cond. Goto - False}} {h, m, \rho, \text{cs}, j}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Procedure Call</strong></th>
<th><strong>Security Cmd</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{cmd}(i) = \text{call } f{e} o$</td>
<td>$\text{cmd}(i) \in SCmd$ $\rho \in \rho_{o}$</td>
</tr>
<tr>
<td>$h, m, \rho, \text{cs}, i$</td>
<td>${h, m, \rho, \text{cs}, i} \xrightarrow{\text{Procedure Call}} {h, m, \rho', \text{cs}', 0}$</td>
</tr>
<tr>
<td>$\rho' = [x_{k} \mapsto v_{k}]_{k=1}^{n}$</td>
<td>$(\rho, \rho_{o}) \leftarrow o$</td>
</tr>
<tr>
<td>$\rho' = [x_{k} \mapsto e_{k}]_{k=1}^{n}$</td>
<td>${h, m, \rho, \text{cs}, i} \xrightarrow{\text{Security Cmd}} {h, m, \rho, \text{cs}, i + 1}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Normal Return</strong></th>
<th><strong>Error Return</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{cmd}(i) = \text{return}$</td>
<td>$\text{cmd}(i) = \text{throw}$</td>
</tr>
<tr>
<td>$\text{cs} = (-, \rho, x, i, j) \cup \text{cs'}$ $\rho(\text{ret}) = v$</td>
<td>$\text{cs} = (-, \rho, x, i, j) \cup \text{cs'}$ $\rho(\text{ret}) = v$</td>
</tr>
<tr>
<td>$o = \text{return}$</td>
<td>$o = \text{return}$</td>
</tr>
<tr>
<td>${h, m, \rho, \text{cs}, i} \xrightarrow{\text{Normal Return}} {h, m, \rho'[x \mapsto v], \text{cs'}, j}$</td>
<td>${h, m, \rho, \text{cs}, i} \xrightarrow{\text{Error Return}} {h, m, \rho'[x \mapsto v], \text{cs'}, j}$</td>
</tr>
</tbody>
</table>

Figure 3.2: Execution for Control Flow Commands: $\{h, m, \rho, \text{cs}, i\} \xrightarrow{\text{cmd}} \{h', m', \rho', \text{cs}', j\}$

Below we give an explanation of the semantics of each command.

**Basic Command**: The control flow semantics executes a basic command using the semantics of basic commands.

**Security Command**: The control flow semantics executes a Security command using the semantics of Security commands.

**Unconditional Goto**: When executing an unconditional goto goto $j$, the control flow semantics simply jumps to the $j$-th command of the current procedure.

**Conditional Goto**: When executing a conditional goto goto $[e] j$, $k$, the semantics first evaluates the guard expression $e$. If $e$ is evaluated to be true, then the program will jump to the $j$-th command of the current procedure; otherwise it will jump to the $k$-th command of the current procedure.

**Procedure Call**: When executing a procedure call $x := e_{k} [n_{k=0}]$ with $j$, the semantics first evaluates the caller expression $e_{k}$, and the arguments expressions $e_{k}[n_{k=0}]$, obtaining $v_{k} = e_{k}[n_{k=0}]$. Then, it creates a new store $\rho'$ for the execution of $f$. Finally, the semantics creates a new call stack entry to keep track of the execution context of the current function, so that control can correctly be returned once the execution of $f$ finishes.

**Return**: When the program reaches the normal return index $i_{nm}$, it obtains the value $v$ associated with the variable $\text{ret}$ in the current store. The program then discards the last entry of the call stack and updates the store by assigning the value $v$ to $x$ (the variable present in the call stack tuple of the form $(f, \rho, x, i, j)$).
Error: When the program reaches the error return index \( i_e \), it obtains the value \( v \) associated with the variable \( \text{ret} \) in the current store. The program then discards the last entry of the call stack and updates the store by assigning the value \( v \) to \( x \) (the variable present in the call stack tuple of the form \( (f, \rho, x, i, j) \)).

### 3.2.3 Security Commands

In this subsection we explain the semantics of the JSIL Security Commands, the data structures used in this subsection will be introduced in the next sections. The semantics of JSIL Security Commands is given in Figure 3.3; semantic transitions have the form \( \{ \rho, sc \} \Downarrow o \), where:

- \( \{ \rho, sc \} \) is the initial configuration consisting of a store and a security command.
- \( o \) represents the event label to be sent to the monitor.

![Figure 3.3: Semantics of Security Commands: \( \{ \rho, sc \} \Downarrow o \)](image)

Below explain the semantics of each command.

**Upgrade Variable**: The upgrade variable command \( \text{upgVar} (x, \sigma) \) upgrades the security level of \( x \) to \( \sigma \).

**Upgrade Property**: The upgrade property command \( \text{upgProp} (e_1, e_2, \sigma) \) first evaluates \( e_1 \) and \( e_2 \), to a location \( l \) and a property \( p \) respectively. Then it updates the value level of \( (l, p) \) to \( \sigma \).

**Upgrade Property-\( \epsilon \)**: The upgrade property-\( \epsilon \) \( \text{upgProp} (e_1, e_2, \sigma) \) first evaluates \( e_1 \) and \( e_2 \), to a location \( l \) and a property \( p \) respectively. Then it updates the existence level of \( (l, p) \) to \( \sigma \).

**Upgrade Structure**: The upgrade structure \( \text{upgStruct} (e, \sigma) \) first evaluates \( e \) to a location \( l \). Then it updates the structure level of \( l \) to \( \sigma \).

**Upgrade Metadata**: The upgrade structure \( \text{upgMtd} (e, \sigma) \) semantics first evaluates \( e \) to a location \( l \). Then it updates the metadata level of \( l \) to \( \sigma \).

**Merge**: The merge \( \text{merge} (i) \) semantics pops the context level.

### 3.3 General Monitored Semantics

In this section we propose a new general monitored semantics for JSIL. The goal of the monitored semantics is to couple the execution of a JSIL program with a parametric small-step monitor that runs in lockstep with the JSIL semantics. The semantics is designed in a way that separates the behaviour of the monitor from that of the JSIL semantics. This leads to a modular implementation and modular
o ::= • | assign \((x, e_v)\) | new \((l, p, e_l, e_p)\) | p-assign \((l, p, e_l, e_p)\) | p-lookup \((x, l, p, e_l, e_p)\) | delete \((l, p, e_l, e_p)\) | hasField \((x, l, p, e_l, e_p)\) | hasField \((x, l, p, e_l, e_p)\) | getFields \((x, l, e_l)\) | metadata \((x, l, e_l)\) | metadata \((x, l, e_l)\) | upgProp \((l, p, e_l, e_p, \sigma)\) | upgProp \((l, p, e_l, e_p, \sigma)\) | upgStruct \((l, e_l, \sigma)\) | upgMtd \((l, e_l, \sigma)\) | upgVar \((x, \sigma)\) | \(\gamma(e, i)\) | \(\alpha(e, \bar{i})\) | \(\text{call} (e, \text{arg}, x)\)

Figure 3.5: Event Labels

The monitored semantics has a single rule, which makes use of the semantics of control flow commands and a given information flow monitor. A configuration of the monitored semantics consists of:

- \(\Omega\), configuration of the JSIL control flow semantics; and
- \(\Phi\), configuration of the monitor.

In order to perform a monitored transition \(\Phi \rightarrow^{\omega} \Phi'\), the monitored semantics first performs a step of the JSIL semantics, \(\Omega \rightarrow^{\omega} \Omega'\), obtaining a new JSIL configuration and an event \(\omega\). Then, it feeds the event \(\omega\) to the monitor semantics, obtaining a new monitor configuration \(\Phi'\). The final monitored semantics configuration consists of the pair \((\Omega', \Phi')\).

The monitored semantics illustrate with the given information flow monitor our labels. These labels are described in Figure 3.5. When a monitor receives a label it processes it in any way it sees fit. This approach abstracts the monitor from the JSIL semantics of JSIL making the change of monitor straightforward. To illustrate this let us consider the following two monitors:

- **Permissive monitor**: This monitor has a single configuration denoted \(\heartsuit\). The monitor has one possible transition for any given label, effectively accepting all possible program steps.

- **Obstructive monitor**: whose monitor configuration consists of \(\spadesuit\). The monitor had no possible transition for any given label, effectively rejecting all possible program steps.
Although these monitors are very limited in the way they interpret the labels they receive, they are a good example to see how easy it is to change information flow monitor and how our generalized monitored semantics separates the semantics of the JSIL language from that of the information flow monitor.

### 3.4 Running Example

In this section, we illustrate how the JSIL semantics works by appealing to the JS running example given in Section 1.2. Figure 3.6 shows a stylised JSIL compilation of the body of the function getRightBit1.

The compiled JSIL code makes use of the following dedicated procedures to capture the semantics of JS:

- **putValue(l, p, v):** assigns the value $v$ to the property $p$ of the object at location $l$.
- **getValue(l, p):** returns the value $v$ assigned to the property $p$ of the object at location $l$. At first glance this procedure is analogous to the `LOOKUP` command, however it couldn’t be further from the truth as this procedure was designed to take into consideration all the possible behaviours of JavaScript, traversing the prototype chain if needed.
- **defineOwnProperty(l, p, d):** updates the value of the property $p$ in location $l$ to be $d$.
- **oRef(l, p):** returns a reference to the value of the property $p$ at location $l$.
- **vDesc(v, b_1, b_2, b_3):** creates a property descriptor denoting a value $v$ with configurability $b_1$, with enumerability $b_2$ and with writability $b_3$, where $b_1, b_2, b_3$ are Boolean values.

Figure 3.6 show a stylised compilation of the JS program given in Figure 1.4. Recall that the mentioned JS program has an information flow leak via the length of the array $a$, which has low security level. More concretely, depending on the parity of the high variable $x$, the program will set the length of the array $a$ to either 1 or 2. Below, we see that the same information flow leak is present in the JSIL version of the program.

In order to understand how this program gets executed by the semantics of JSIL, we present the corresponding execution trace in Figures 4.12 and 4.14. To avoid clutter, we only show the heap and the store components of JSIL configurations. The program starts by creating the array $a$ and by populating it (lines 0-3). Then, it checks whether or not $x$ is even (line 4).

If $x$ is not even the control jumps to line 8, otherwise the program creates a property descriptor with configurability `false` (line 5) and assigns it to the first position of the array (line 6). Then, the control jumps to line 8, where both branches of the `goto` merge.

Now, the program creates a reference $aRef$ to the length of the array $a$ (line 8) and uses it to set the length of the array to 1 (line 9). Finally, the program checks the value of the current length of the array (line 10) and depending on whether or not it is 1, it return 1 (lines 12-13) or 0 (lines 14-15).

The information flow leak present in 1.4 still exists, as can be seen by running the program with an odd input. In the next section we will present an information flow monitor that precludes all types of information flow leaks.

22
Figure 3.7: Running example when x is odd
Figure 3.8: Running example when x is even
```javascript
a := initArray()
putValue(a,'0','a')
putValue(a,'1','b')
putValue(a,'2','c')
goto[(x%2=0)] then1 next1
then1: desc := vDesc('b', false, true, true)
defineOwnProperty(a, '1', desc)
goto next1
next1: aRef := oRef(a, 'length')
putValue(aRef, 1)
y := getValue(aRef)
goto[y=1] then2 else2
then2: ret := 1
return
else2: ret := 0
return
```

Figure 3.6: Running Example in JSIL
Chapter 4

JSIL Monitor

In this chapter we will first explain the information flow labellings used to secure JSIL sets and their meaning (Section 4.1). Afterwards we analyse all the possible leaks for the JSIL language (Section 4.2). Taking this information into consideration, we present a information flow monitor to prevent the identified leaks (Section 4.3). Finally, we will analyse how the presented monitor executes the running example (Section 4.4).

4.1 JSIL Security

The information flow monitor presented in this section helps track changes with the security labels associated with the different type of resources existing in JSIL stacks (object properties, metadata and variables). This monitor follows the no sensitive upgrade discipline (henceforth called NSU-discipline). We do this via the security domains described in the table below, respectively corresponding to the security heap, security metadata and security store. In the table below, we formally define the JSIL security domains: security heaps, security metadata, and security store. Below, we explain the structure of these security domains.

**JSIL Security Domains**

| Security Heap: | $sh \in S\text{Heap} :: \text{Locs} \times \text{Props} \rightarrow \text{Lev} \times \text{Lev} |
| Security Metadata: | $sm \in S\text{Metadata} :: \text{Loc} \rightarrow \text{Lev} \times \text{Lev}$ |
| Security Store: | $sp \in S\text{Store} :: \text{Var} \rightarrow \text{Lev}$ |

- **Security heaps**: A security heap $sh \in S\text{Heap}$ is a partial function mapping pairs consisting of a location $l \in \text{Locs}$ and property $p \in \text{Props}$ to two security levels. If $sh(l, p) = (\sigma_e, \sigma_p)$, then $\sigma_e$ represents the existence level of property $p$ in the object at location $l$ and $\sigma_p$ represents its property level. The existence level is an upper bound on the levels of the contexts in which the property $p$ can be created or deleted. The property level corresponds to the security level of the value stored inside that property.
Intuitively, if \( \text{of an attacker at a given security level } \sigma \)

\[ h \] cells; formally

\[
\text{Empty heap:}
\]

If the heap is empty

\[ h \]

\[ \text{Non-empty heap:} \]

In order to compute the low projection of a non-empty heap, we partition the given heap into a singleton heap, containing a single heap cell, and a heap \( h' \) containing all the remaining cells; formally \( h = h' \cup (l,p) \rightarrow v \). Then, we partition the security heap \( sh \) in the exact same way,
sh = sh′ ⊔ (l, p) → (σ_e, σ_p), obtaining the existence level, σ_e, and the property level, σ_p, of the heap cell (l, p) → v as well as the security heap sh′ corresponding to the heap h′. Using sh′, we recursively compute the low projection of h′ at level σ obtaining the values of the heap h, that an observer at level σ can see. Now there are three possible cases, each corresponding to a separate rule in Figure 4.1.

- **Visible Cell with Visible Value:** In this case both σ_e and σ_p are less than or equal to σ, the observation level of the attacker. Hence, the attacker can see both the existence of the cell and the value that it contains. Accordingly, the projection returns the cell as unioned to a projection of the rest of the heap.

- **Visible Cell with Invisible Value:** In this case σ_e is less or equal to σ, but σ_p is greater than σ, where σ is the observation level of the attacker. Hence, the attacker can see the existence of the cell, but does now know what value it contains. Accordingly, the projection returns the cell as unioned to a projection of the rest of the heap.

- **Invisible Cell with Invisible Value:** In this case both σ_e and σ_p are greater than σ, where σ is the observation level of the attacker. Hence, the attacker cannot see the existence of the cell. Accordingly, the projection returns a projection of the rest of the heap.

In order to exemplify how the low projection works, we show how to apply it to the heap given in Figure 4.1. Consider a lattice with only two security levels L (low) and H (high) and that the observation level σ is L. Figure 4.1 represents a JSIL heap containing 3 objects, respectively at locations l_1, l_2 and l_3. The object at location l_1 contains two properties, a and b, where the value of a is the location l_2 and the value of b is the location l_3. The object at location l_2 has only the property c whose value is 3. Finally, the object at location l_3 has two properties, d and e, whose values are 4 and "xpto" respectively.

Figure 4.2 shows the security levels (existence level and value level) associated with each property in the secure heap. For instance, property c in the object at location l_2 is mapped to the pair (H, H), meaning the it has a high existence level and an high value level.

Figure 4.3 depicts the result of applying the low projection function to the heap given in Figure 4.1 labelled as in Figure 4.2. In the object at location l_1 the observer knows that the property a exists but cannot see its value, while he can see both the value and existence of property b. In the object at location l_2 the observer cannot see any property. Finally, in the object at location l_3 the observer can only see the property e and knows that its value is "xpto".

### 4.2 JSIL Information Flow Leaks

The monitor to be presented follows, the no sensitive upgrade discipline, in a nutshell this means that visible resources cannot be upgraded in invisible contexts. For instance consider the program **Type I** in Table 4.2 and that the variable h has an high security level. The program starts by initializing the variables l and laux, these variables will be labelled low as they were created in a low context. As the program enters the goto in line 3, the context level will increase to the level of the guard, in this case
Figure 4.1: Example JSIL heap

Figure 4.2: Example monitored JSIL security heap
Figure 4.3: Low projection for a low level observer

$h$. Now, considering the value of $h$ to be true the assign to $laux$ would be stopped by the no sensitive upgrade discipline, as this assign would leak information about $h$, which has a high security level.

Below, we identify all the possible types of sensitive upgrades that exist in JSIL, showing how they can be exploited to encode information flow leaks. As in [33] we have divided all leaks into 7 different types. Table 4.2 shows all the leaks.

- **Visible Variable Assignment in an Invisible Context (Type I):** the monitor blocks assignments to variables holding visible values in high contexts. Therefore, in the example, the monitor blocks the assignment of $false$ to $laux$ inside the first conditional.

- **Visible Property Assignment in an Invisible Context (Type II):** the monitor blocks assignments to properties holding visible values within invisible contexts. Therefore, in the example, the monitor blocks the assignment of $false$ to $o.p$ inside the first conditional.

- **Visible Property Deletion in an Invisible Context (Type III):** the monitor blocks deletions of visible properties in invisible contexts. Therefore, in the example, the monitor blocks the deletion of the property "$p$" of the object bound to $o$ inside the first conditional.

- **Visible Property Assignment via Invisible Location (Type IV):** the monitor blocks assignments to visible properties when the location pointing to the object that binds the property was computed using secret information. For instance, in the example, while the low variable $o1$ can only hold low locations, the high variable $oh$ can hold both low and high locations. Therefore, the assignment $oh = o1$ is allowed to go through. However, when $oh$ is set to point to the same location as $o1$, the
**Table 4.2: Naive Approach vs No Sensitive Upgrade**

<table>
<thead>
<tr>
<th>Type I</th>
<th>Type II</th>
<th>Type III</th>
<th>Type IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>then</td>
<td>goto</td>
<td>mg2</td>
</tr>
<tr>
<td>laux</td>
<td>true</td>
<td>laux</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>goto</td>
<td>mg1</td>
<td>skip</td>
</tr>
<tr>
<td>th</td>
<td>mg1</td>
<td>th1</td>
<td>mg2</td>
</tr>
<tr>
<td>th1</td>
<td>mg1</td>
<td>laux</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>mg2</td>
<td>skip</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>p</td>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
<td>proph</td>
<td>proph</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>mg2</td>
<td>skip</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>p</td>
<td>proph</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
<td>proph</td>
<td>proph</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>mg2</td>
<td>skip</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type V</th>
<th>Type VI</th>
<th>Type VII</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>then</td>
<td>goto</td>
</tr>
<tr>
<td>mg1</td>
<td>true</td>
<td>mg2</td>
</tr>
<tr>
<td>th1</td>
<td>mg1</td>
<td>th2</td>
</tr>
<tr>
<td>th2</td>
<td>mg2</td>
<td>laux</td>
</tr>
<tr>
<td>mg2</td>
<td>skip</td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>new</td>
<td>o</td>
</tr>
<tr>
<td>p</td>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
<td>proph</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>mg2</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
<td>o</td>
</tr>
<tr>
<td>p</td>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
<td>proph</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>mg2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type VIII</th>
<th>Type IX</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>then</td>
</tr>
<tr>
<td>mg1</td>
<td>true</td>
</tr>
<tr>
<td>th1</td>
<td>mg1</td>
</tr>
<tr>
<td>mg1</td>
<td>skip</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type X</th>
<th>Type XI</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>then</td>
</tr>
<tr>
<td>mg1</td>
<td>true</td>
</tr>
<tr>
<td>th1</td>
<td>mg1</td>
</tr>
<tr>
<td>mg1</td>
<td>skip</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type XII</th>
<th>Type XIII</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>then</td>
</tr>
<tr>
<td>mg1</td>
<td>true</td>
</tr>
<tr>
<td>th1</td>
<td>mg1</td>
</tr>
<tr>
<td>mg1</td>
<td>skip</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type XIV</th>
<th>Type XV</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>then</td>
</tr>
<tr>
<td>mg1</td>
<td>true</td>
</tr>
<tr>
<td>th1</td>
<td>mg1</td>
</tr>
<tr>
<td>mg1</td>
<td>skip</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type XVI</th>
<th>Type XVII</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>then</td>
</tr>
<tr>
<td>mg1</td>
<td>true</td>
</tr>
<tr>
<td>th1</td>
<td>mg1</td>
</tr>
<tr>
<td>mg1</td>
<td>skip</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type XVIII</th>
<th>Type XIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>then</td>
</tr>
<tr>
<td>mg1</td>
<td>true</td>
</tr>
<tr>
<td>th1</td>
<td>mg1</td>
</tr>
<tr>
<td>mg1</td>
<td>skip</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type XX</th>
<th>Type XXI</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>then</td>
</tr>
<tr>
<td>mg1</td>
<td>true</td>
</tr>
<tr>
<td>th1</td>
<td>mg1</td>
</tr>
<tr>
<td>mg1</td>
<td>skip</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type XXII</th>
<th>Type XXIII</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>then</td>
</tr>
<tr>
<td>mg1</td>
<td>true</td>
</tr>
<tr>
<td>th1</td>
<td>mg1</td>
</tr>
<tr>
<td>mg1</td>
<td>skip</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>o</td>
<td>new</td>
</tr>
<tr>
<td>p</td>
<td>proph</td>
</tr>
<tr>
<td>proph</td>
<td>q</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>
assignment \( \text{oh} \cdot \text{p} = \text{true} \) is blocked, since it tries to update the value of a \textit{low} property via a \textit{high} location.

- **Visible Property Assignment via an Invisible Property Name (Type V):** the monitor blocks assignments to visible properties when the corresponding property name was computed using secret information. For instance, in the example, the variable \( \text{proph} \) can hold both \textit{low} and \textit{high} property names. Therefore, the assignment \( \text{proph} = \text{"q"} \) is allowed to go through, even though it is performed inside a \textit{high} conditional. However, after this assignment, the assignment \( \text{o} \left[ \text{proph} \right] = \text{true} \) is blocked since it tries to update the value of a \textit{low} property via a \textit{high} property name.

- **Visible Property Deletion via an Invisible Location (Type VI):** the monitor blocks the deletion of visible properties when the location pointing to the object that binds the property was computed using secret information. For instance, in the example, the \textit{high} variable \( \text{oh} \) can hold both \textit{low} locations and \textit{high} locations. Therefore, the assignment \( \text{oh} = \text{ol} \) is allowed to go through. However, when \( \text{oh} \) is set to point to the same location as \( \text{ol} \), the execution of \( \text{delete oh} \) is blocked since it constitutes a \textit{low} property deletion via a \textit{high} location.

- **Visible Property Deletion via an Invisible Property Name (Type VII):** the monitor blocks the deletion of a visible property when the corresponding property name was computed using secret information. For instance, in the example, the \textit{high} variable \( \text{proph} \) can hold both \textit{low} property names and \textit{high} property names. Therefore, the assignment \( \text{proph} = \text{"p"} \) is allowed to go through inside the body of the \textit{high} conditional. However, when \( \text{proph} \) is set to \text{"p"}, the execution of \( \text{delete proph} \) is blocked since it constitutes a \textit{low} property deletion via a \textit{high} property name.

### 4.3 An Information Flow Monitor for JSIL

The semantics of our information flow monitor is given in Figures 4.4 and 4.5 in a small-step style; semantic transitions have the form \( \{sh, sm, sp, scs, pc\} \xrightarrow{o_{\text{mon}}} \{sh', sm', sp', scs', pc'\} \), where:

- \( \{sh, sm, sp, scs, pc\} \) is the initial monitor configuration, consisting of a security heap \( sh \in S\text{Heap} \), a security metadata table \( m \in S\text{Metadata} \), a security store \( sp \in S\text{Store} \), a security call stack \( scs \in S\text{Call} \), and a security program counter \( pc \in PC \);

- \( o_{\text{mon}} \) represents the monitor transition the label \( o \) sent by the program;

- \( \{sh', sm', sp', scs', pc'\} \) is the final monitor configuration consisting of a security heap, a security metadata table, a security store, a security call stack, and a security program counter.

The table Below shows the syntax of security call stacks \( scs \), security program counter \( pc \), monitor configuration \( \Phi \), monitor transition \( \Phi \xrightarrow{o_{\text{mon}}} \Phi' \). The security heap \( sh \), security metadata table \( sm \), security store \( sp \) have already been discussed in Section 4.1. The security call stack \( scs \) is a list of tuples of the form \( (x, pc, sp) \), where: \( x \) is the variable where the execution will return to, \( pc \) is the program counter at the time of the call, \( sp \) is the security store at the time of the call. Intuitively, the security call stack
If the label received by the monitor is of type `upgVar`, to ease the interpretation of the monitor rules where: red is used to highlight the constraints of a given rule, orange is used to highlight updates to the monitor configuration.

**Upgrade Variable**

\[
o = \text{upgVar}(x, \sigma) \quad \text{lev}(pc) \sqsubseteq sp(x)
\]

\[
\{ sh, sm, sp, scs, pc \} \xrightarrow{o \text{ mon}} \{ sh, sm, sp[x \mapsto \sigma \cup \text{lev}(pc)], scs, pc \}
\]

**Upgrade Property**

\[
o = \text{upgProp}(l, p, e_1, e_p, \sigma)
\]

\[
\sigma = sh \cup (l, p) \mapsto (\sigma_e, \sigma_c)
\]

\[
\vec{o} = \text{lev}(sp, e_1) \sqcup \text{lev}(sp, e_p) \sqcup \text{lev}(pc) \sqsubseteq \sigma'
\]

\[
sh'' \equiv sh' \cup (l, p) \mapsto (\sigma_e, \sigma_c \sqcup \sigma)
\]

\[
\{ sh, sm, sp, scs, pc \} \xrightarrow{o \text{ mon}} \{ sh'', sm, sp, scs, pc \}
\]

**Upgrade Property**

\[
o = \text{upgProp}(l, p, e_1, e_p, \sigma)
\]

\[
\sigma = sh \cup (l, p) \mapsto (\sigma_e, \sigma_c)
\]

\[
\vec{o} = \text{lev}(sp, e_1) \sqcup \text{lev}(sp, e_p) \sqsubseteq \sigma'
\]

\[
sh'' = sh' \cup (l, p) \mapsto (\sigma_e \sqcup \sigma_c)
\]

\[
\{ sh, sm, sp, scs, pc \} \xrightarrow{o \text{ mon}} \{ sh'', sm, sp, scs, pc \}
\]

**Upgrade MetaData**

\[
o = \text{upgMtd}(l, e, \sigma)
\]

\[
sm = sm' \sqcup l \mapsto (\sigma_s, \sigma_m)
\]

\[
\vec{o} = \text{lev}(sp, e) \sqcup \text{lev}(pc) \sqsubseteq \sigma_m
\]

\[
sm'' = sm' \sqcup l \mapsto (\sigma_s, \sigma_e \sqcup \sigma)
\]

\[
\{ sh, sm, sp, scs, pc \} \xrightarrow{o \text{ mon}} \{ sh'', sm'', sp, scs, pc \}
\]

**Upgrade Structure**

\[
o = \text{upgStruct}(l, e, \sigma)
\]

\[
sm = sm' \sqcup l \mapsto (\sigma_s, \sigma_m)
\]

\[
\vec{o} = \text{lev}(sp, e) \sqcup \text{lev}(pc) \sqsubseteq \sigma_s
\]

\[
sm'' = sm' \sqcup l \mapsto (\sigma_e \sqcup \sigma_s, \sigma_m)
\]

\[
\{ sh, sm, sp, scs, pc \} \xrightarrow{o \text{ mon}} \{ sh'', sm'', sp, scs, pc \}
\]

Figure 4.4: Monitored Semantics Upgrades: \{sh, sm, sp, scs, pc\} \xrightarrow{o \text{ mon}} \{sh', sm', sp', scs', pc'\}

keeps track of the monitor execution context of the current function, so that the monitor state can be correctly updated once the current function finishes execution. The program counter `pc` is a list of pairs of the form \((\sigma, i)\) where \(\sigma\) is a security level and \(i\) command index. Intuitively if \((\sigma, i) \in pc\), then the current execution branched in the command with index \(i\) depending in a value with a security level \(\sigma\). As explained previously the monitor configuration consists of \{sh, sm, sp, scs, pc\} and the monitor transition represents the change in the monitor configuration triggered by the label \(o\).

**JSIL Monitor Structures**

<table>
<thead>
<tr>
<th>Security Call Stack:</th>
<th>(scs \in SCS :: (x, pc, sp) :: scs )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security Program Counter:</td>
<td>(pc \in PC :: (\sigma, i) :: pc )</td>
</tr>
<tr>
<td>Monitor Configuration:</td>
<td>(\Phi ::= {sh, sm, sp, scs, pc} )</td>
</tr>
<tr>
<td>Monitor Transition:</td>
<td>(\Phi \xrightarrow{o \text{ mon}} \Phi')</td>
</tr>
</tbody>
</table>

### 4.3.1 Monitor Semantics of Upgrade Commands

Below, we explain the semantics of the upgrade commands given in Figure 4.4. We use a color code to ease the interpretation of the monitor rules where: red is used to highlight the constraints of a given rule, orange is used to highlight updates to the monitor configuration.

**Upgrade Variable**

\[
o = \text{upgVar}(x, \sigma) \quad \text{lev}(pc) \sqsubseteq sp(x)
\]

\[
sp' = sp[x \mapsto \sigma \cup \text{lev}(pc)]
\]

\[
\{ sh, sm, sp, scs, pc \} \xrightarrow{o \text{ mon}} \{ sh, sp', scs, pc \}
\]

If the label received by the monitor is of type `upgVar(x, \sigma)`, and the security level of the program counter is lower or equal to the security level of \(x\) (\(\text{lev}(pc) \sqsubseteq sp(x)\)) the security level of \(x\) is updated to the least upper bound of the level of the program counter and the the level of \(\sigma\).
Upgrade Property:

\[ a = \text{upgProp} (l, p, e_l, e_p, \sigma) \quad s_h = s_h' \uplus (l, p) \mapsto (\sigma_e, \sigma') \]

\[ \hat{\sigma} = \text{lev}(sp, e_l) \cup \text{lev}(sp, e_p) \subseteq \sigma' \quad s_h'' = s_h' \uplus (l, p) \mapsto (\sigma_e, \hat{\sigma} \sqcup \sigma) \]

\[ \{s_h, sm, sp, scs, pc\} \xrightarrow{a \; \text{mon}} \{s_h'', sm, sp, scs, pc\} \]

If the label received by the monitor is of type \text{upgProp}(l, p, e_l, e_p, \sigma), the monitor first searches the security heap for the value level \sigma' and the existence level \sigma_e of the property \(p\) of the object at location \(l\). The monitor then computes the least upper bound between the level of the program counter (\text{lev}(pc)), the level of \(e_l\) (\text{lev}(sp, e_l)), and the level of \(e_p\) (\text{lev}(sp, e_p)), assigning the result to \(\hat{\sigma}\). Following the NSU-discipline, the upgrade is only allowed to take place if the \(\hat{\sigma}\) is lower than or equal to \(\sigma'\) (the current value level of \(p\)). Finally, the monitor updates value level of \(p\) in the security heap to the the least upper bound between \(\hat{\sigma}\) and \(\sigma\).

Upgrade Property \(\epsilon\):

\[ a = \text{upgProp}_\epsilon (l, p, e_l, e_p, \sigma) \quad s_h = s_h' \uplus (l, p) \mapsto (\sigma', \sigma_e) \]

\[ \hat{\sigma} = \text{lev}(sp, e_l) \cup \text{lev}(sp, e_p) \subseteq \sigma' \quad s_h'' = s_h' \uplus (l, p) \mapsto (\hat{\sigma} \sqcup \sigma, \sigma_e) \]

\[ \{s_h, sm, sp, scs, pc\} \xrightarrow{a \; \text{mon}} \{s_h'', sm, sp, scs, pc\} \]

If the label received by the monitor is of type \text{upgProp}_\epsilon(l, p, e_l, e_p, \sigma), the monitor first searches the security heap for the value level \sigma_e and the existence level \sigma' of the property \(p\) of the object at location \(l\). The monitor then computes the least upper bound between the level of the program counter (\text{lev}(pc)), the level of \(e_l\) (\text{lev}(sp, e_l)), and the level of \(e_p\) (\text{lev}(sp, e_p)), assigning the result to \(\hat{\sigma}\). Following the NSU-discipline, the upgrade is only allowed to take place if the \(\hat{\sigma}\) is lower than or equal to \(\sigma'\), the current existence level of \(p\). Finally, the monitor updates value level of \(l\) in the security heap to the the least upper bound between \(\hat{\sigma}\) and \(\sigma\).

Upgrade Structure:

\[ a = \text{upgStruct} (l, e, \sigma) \quad sm = sm' \uplus l \mapsto (\sigma_s, \sigma_m) \]

\[ \hat{\sigma} = \text{lev}(sp, e) \cup \text{lev}(pc) \subseteq \sigma_s \quad sm'' = sm' \uplus l \mapsto (\hat{\sigma} \sqcup \sigma, \sigma_m) \]

\[ \{s_h, sm, sp, scs, pc\} \xrightarrow{a \; \text{mon}} \{s_h'', sm, sp, scs, pc\} \]

If the label received by the monitor is of the type \text{upgStruct}(l, e, \sigma) the monitor first searches the security metadata for the security levels associated with the location \(l\). The monitor then computes the least upper bound between the level of the program counter (\text{lev}(pc)), the level of \(e\) (\text{lev}(sp, e)), assigning the result to \(\hat{\sigma}\). Following the NSU-discipline, the upgrade is only allowed to take place if the \(\hat{\sigma}\) is lower than or equal to \(\sigma_s\) (the current structure level of \(l\)). Finally, the monitor updates the structure level of \(l\) in the security metadata to the least upper bound between \(\hat{\sigma}\) and \(\sigma\).

Upgrade Metadata:

\[ a = \text{upgMtd} (l, e, \sigma) \quad sm = sm' \uplus l \mapsto (\sigma_s, \sigma_m) \]

\[ \hat{\sigma} = \text{lev}(sp, e) \cup \text{lev}(pc) \subseteq \sigma_m \quad sm'' = sm' \uplus l \mapsto (\sigma_s, \hat{\sigma} \sqcup \sigma) \]

\[ \{s_h, sm, sp, scs, pc\} \xrightarrow{a \; \text{mon}} \{s_h'', sm, sp, scs, pc\} \]
Silent
\[ \{sh, sm, sp, scs, pc\} \xrightarrow{o_{\text{mon}}} \{sh, sm, sp, scs, pc\} \]

Property assignment 1
\[ o = p\text{-assign}(l, p, e_l, e_p) \]
\[ \begin{align*}
  &\quad \delta = \text{lev}(sp, e_l) \cup \text{lev}(sp, e_p) \cup \text{lev}(pc) \cup \sigma \\
  &\quad \text{sh} = \text{sh}' \cup \{l, p\} \rightarrow (\sigma, -) \cup \text{sh}' \\
  &\quad \{sh, sm, sp, scs, pc\} \xrightarrow{o_{\text{mon}}} \{sh', sm, sp, scs, pc\}
\end{align*} \]

ASSIGNMENT
\[ o = \text{assign}(x, e) \]
\[ \begin{align*}
  &\quad \text{lev}(pc) \subseteq \text{sp}(e) \rightarrow (\text{sv}(x) \cup \text{lev}(pc)) \cup \text{sp}(x) \\
  &\quad \{sh, sm, sp, scs, pc\} \xrightarrow{o_{\text{mon}}} \{sh', sm, sp, [x \mapsto \sigma], scs, pc\}
\end{align*} \]

Property Lookup
\[ o = p\text{-lookup}(x, l, p, e_l, e_p) \]
\[ \begin{align*}
  &\quad \text{sh} = \text{sh}' \cup \{l, p\} \rightarrow (\sigma, -) \cup \text{sh}' \\
  &\quad \delta = \text{lev}(sp, e_l) \cup \text{lev}(sp, e_p) \cup \text{lev}(pc) \cup \text{sp}(x) \\
  &\quad \{sh, sm, sp, scs, pc\} \xrightarrow{o_{\text{mon}}} \{sh', sm, sp, [x \mapsto \sigma], scs, pc\}
\end{align*} \]

HASFIELD - TRUE
\[ o = \text{hasField}(x, l, p, e_l, e_p) \]
\[ \begin{align*}
  &\quad \text{sh} = \{l, p\} \rightarrow (\sigma, -) \cup \text{sh}' \\
  &\quad \delta = \text{lev}(sp, e_l) \cup \text{lev}(pc) \cup \text{sp}(x) \\
  &\quad \{sh, sm, sp, scs, pc\} \xrightarrow{o_{\text{mon}}} \{sh', sm, sp, [x \mapsto \sigma], scs, pc\}
\end{align*} \]

GET FIELDS
\[ o = \text{getFields}(x, l, e_p) \]
\[ \begin{align*}
  &\quad \text{sh} = \{l, p\} \rightarrow (\sigma, -) \cup \text{sh}' \\
  &\quad \delta = \text{lev}(sp, e_l) \cup \text{lev}(pc) \cup \text{sp}(x) \\
  &\quad \{sh, sm, sp, scs, pc\} \xrightarrow{o_{\text{mon}}} \{sh', sm, sp, [x \mapsto \sigma], scs, pc\}
\end{align*} \]

BRANCH
\[ o = (c, i) \]
\[ \text{lev}(sp, e) = \sigma \]
\[ \{sh, sm, sp, scs, pc\} \xrightarrow{o_{\text{mon}}} \{sh, sm, sp, scs, (\sigma, i) : pc\} \]

Figure 4.5: Monitored Semantics:
\[ \{sh, sm, sp, scs, pc\} \xrightarrow{o_{\text{mon}}} \{sh', sm', sp', scs', pc'\} \]

If the label received by the monitor is of type \text{upgMtd}(l, e, \sigma), the monitor first searches the security metadata for the structure level \sigma_s and the metadata level \sigma_m of the location \text{l}. The monitor then computes the least upper bound between the level of the program counter (\text{lev}(pc)), the level of \text{e} (\text{lev}(sp, e)), assigning the result to \delta. Following the NSU-discipline, the upgrade is only allowed to take place if the \delta is lower than or equal to \sigma_m (the current metadata level of \text{l}). Finally, the monitor updates metadata level of \text{l} in the security metadata to the the least upper bound between \delta and \sigma_s.

4.3.2 Semantics of Control Flow Commands

Below we explain the monitor semantics of the control flow commands given in Figure 4.5. We use the color code explained previously to ease the interpretation of the monitor rules.

Silent:
\[ \{sh, sm, sp, scs, pc\} \xrightarrow{o_{\text{mon}}} \{sh, sm, sp, scs, pc\} \]

If the label received by the monitor is of type \text{•*}, the monitor does not do anything, as this label means that there is no need for the monitor to take any action since the command being executed is unrelated with leaks.
Object Creation:

\[ o = \text{new}(x, l, e) \quad \sigma_{pc} = \text{lev}(pc) \quad sm' = sm \uplus l \mapsto (\sigma, \sigma) \quad \sigma = \text{lev}(sp, e) \sqcup \sigma_{pc} \]

\[ \{sh, sm, sp, scs, pc\} \xrightarrow{\text{mon}} \{sh, sm', sp[x \mapsto \sigma], scs, pc\} \]

If the label received by the monitor is of type \( \text{new}(x, l, e) \), the monitor first assigns the level of the program counter \( \text{lev}(pc) \) to the meta variable \( \sigma_{pc} \), then, the monitor assigns the value of the least upper bound between \( \sigma_{pc} \) and the level of \( e \) under the secure store \( \text{lev}(sp, e) \) to the meta variable \( \sigma \). Intuitively, \( \sigma \) corresponds to the level of the object being created. Then, the monitor creates a new entry in the security metadata table assigning the newly created object to \( \sigma \) (\( \sigma \) is both used as the structure security level of the object and the metadata level of the object). Finally, the monitor updates the level of \( x \) in the security store to \( \sigma \).

Property Assign 1:

\[ o = \text{p-assign}(l, p, e_t, e_r, e_v) \quad sh = (l, p) \mapsto (\sigma_t, \sigma) \uplus sh' \]

\[ \hat{\sigma} = \text{lev}(sp, e_t) \sqcup \text{lev}(sp, e_r) \sqcup \text{lev}(pc) \sqcup \sigma \quad sh'' = sh' \uplus (l, p) \mapsto (\sigma_t, \hat{\sigma} \sqcup \text{lev}(sp, e_v)) \]

\[ \{sh, sm, sp, scs, pc\} \xrightarrow{\text{mon}} \{sh'', sm, sp, scs, pc\} \]

If the label received by the monitor is of type \( \text{p-assign}(l, p, e_t, e_r, e_v) \), the monitor first searches the security heap for the value level and the existence level of the property \( p \) (\( \sigma \) and \( \sigma_e \) respectively). The monitor then computes the least upper bound between the level of the program counter, \( \text{lev}(pc) \), the level of \( e_t \), \( \text{lev}(sp, e_t) \), and the level of \( e_r \) \( \text{lev}(sp, e_r) \), assigning the result to \( \hat{\sigma} \). Following the NSU-discipline, the upgrade is only allowed to take place if the \( \hat{\sigma} \) is lower than or equal to \( \sigma \) (the current value level of \( p \)). Finally, the monitor updates value level of \( p \) in the security heap to the the least upper bound between \( \hat{\sigma} \) and \( \text{lev}(sp, e_v) \), the security level of the value being assigned.

Property Assign 2:

\[ o = \text{p-assign}(l, p, e_t, e_r, e_v) \quad (l, p) \notin \text{dom}(sh) \quad sm = l \mapsto (\sigma_t, -) \uplus - \]

\[ \hat{\sigma} = \text{lev}(sp, e_t) \sqcup \text{lev}(sp, e_r) \sqcup \text{lev}(pc) \sqcup \sigma \quad sh = sh' \uplus (l, p) \mapsto (\sigma_t, \hat{\sigma} \sqcup \text{lev}(sp, e_v)) \]

\[ \{sh, sm, sp, scs, pc\} \xrightarrow{\text{mon}} \{sh', sm, sp, scs, pc\} \]

If the label received by the monitor is of type \( \text{p-assign}(l, p, e_t, e_r, e_v) \), the monitor first searches the security heap for the existence level and the value level of \( p \). Given that \( p \) is not present in the security heap, the monitor obtains the structure level \( \sigma_s \) of \( l \) from the security metadata \( sm \). Then, it computes the least upper bound between the level of the program counter \( \text{lev}(pc) \), the level of \( e_t \), \( \text{lev}(sp, e_t) \), and the level of \( e_r \), \( \text{lev}(sp, e_r) \), assigning the result to \( \hat{\sigma} \). Following the NSU-discipline, the upgrade is only allowed to take place if \( \hat{\sigma} \) is lower than or equal to \( \sigma_s \) (the current structure level of \( l \)). Finally, the monitor creates a new entry for property \( p \) in the security heap, associating \( p \) with existence level \( \hat{\sigma} \) and value level least upper bound between \( \hat{\sigma} \) and \( \text{lev}(sp, e_v) \).

Assign:

\[ o = \text{assign}(x, e) \quad \text{lev}(pc) \subseteq sp(x) \quad \sigma = \text{lev}(sp, e) \sqcup \text{lev}(pc) \quad sp' = sp[x \mapsto \sigma] \]

\[ \{sh, sm, sp, scs, pc\} \xrightarrow{\text{mon}} \{sh, sm, sp', scs, pc\} \]
If the label received by the monitor is of type assign \((x, e)\), meaning that the value of the expression \(e\) is being assigned to variable \(x\). The monitor only allows the assignment if the level of the program counter \(\text{lev}(pc)\) is lower or equal than the current level of the variable \(s_p(x)\). If the condition is met, the monitor changes the security level of \(x\) \((\sigma)\) to the least upper bound between the level of the program counter and the level of the expression \(\text{lev}(s_p, e)\). Finally the monitor updates the security level of \(x\) in the store to \(\text{lev}(s_p, e)\).

**Call:**

\[
o = \text{call} (e, [x_i : e_i]_{i=1}^{n}, x) \quad \sigma_f = \text{lev}(s_p, e) \quad \hat{\sigma} = \text{lev}(pc) \sqcup \sigma_f \\
\sigma_i = \text{lev}(s_p, e_i) \sqcup \text{lev}(pc) \mid_{i=1}^{n} \quad \text{scs}' = \{x, \text{pc}, s_p\} :: \text{scs} \quad s_p' = \{x_i \mapsto \sigma_i\}_{i=1}^{n}
\]

If the label received by the monitor is of type call \((e, [x_i : e_i]_{i=1}^{n}, x)\), the monitor first assigns \(\text{lev}(s_p, e)\) to the variable \(\sigma_f\); intuitively \(\sigma_f\) is the security level of the identifier of the procedure being called. The monitor then assigns the least upper bound between \(\text{lev}(pc)\) and \(\sigma_f\) to \(\hat{\sigma}\). Afterwards, the monitor computes the security level of every formal parameter of the procedure being called. The level of the \(i\)-th parameter \((x_i)\) corresponds to the lub between the level of the corresponding argument \((\text{lev}(s_p, e_i))\) and the current pc \((\text{lev}(pc))\). These levels are used to construct the security store under which the procedure executes, \(s_p'\). Afterwards, the monitor updates the call stack with \((x, \text{pc}, s_p) :: \text{scs}\) which will be used to re-instantiate the security context when the procedure returns.

**Lookup:**

\[
o = \text{p-lookup} (x, l, p, e_l, e_p) \quad s_h = s_h' \sqcup (l, p) \mapsto (-, \sigma_v) \quad \hat{\sigma} = \text{lev}(s_p, e_l) \sqcup \text{lev}(s_p, e_p) \sqcup \text{lev}(pc) \sqsubseteq s_p(x)
\]

If the label received by the monitor is of type \(\text{p-lookup} (x, l, p, e_l, e_p)\), the monitor first searches the security heap for the value level of \(p\). Then, the monitor computes the least upper bound between the level of the program counter, \(\text{lev}(pc)\), the level of \(e_l\), \(\text{lev}(s_p, e_l)\), and the level of \(e_p\), \(\text{lev}(s_p, e_p)\), assigning the result to \(\hat{\sigma}\). Following the NSU-discipline, the upgrade is only allowed to take place if \(\hat{\sigma}\) is lower or equal to \(s_p(x)\) (the security level of \(x\)). Finally the monitor updates value of \(x\) to \(\hat{\sigma} \sqcup \sigma_v\) in the security store.

**Delete:**

\[
o = \text{delete} (l, p, e_l, e_p) \quad s_m = l \mapsto (\sigma_s, -) \sqcup - \quad s_h = (l, p) \mapsto (\sigma_v, -) \sqcup s_h' \quad \hat{\sigma} = \text{lev}(s_p, e_l) \sqcup \text{lev}(s_p, e_p) \sqcup \text{lev}(pc) \sqsubseteq \sigma_s \sqcap \sigma_v
\]

If the label received by the monitor is of type delete \((l, p, e_l, e_p)\), the monitor first searches the security metadata for the structure level of \(l\). The monitor then searches the security store for the existence level of the property \(p\). The monitor then computes the least upper bound between the level of the program counter \(\text{lev}(pc)\), the level of \(e_l\), \(\text{lev}(s_p, e_l)\), and the level of \(e_p\), \(\text{lev}(s_p, e_p)\), assigning the result to
Following the NSU-discipline, the upgrade is only allowed to take place if the \( \hat{\sigma} \) is lower than or equal to the greatest lower bound between \( \sigma_s \) and \( \sigma_e \).

**Has Field - True:**

\[
o = \text{hasField}(x, l, p, e_l, e_p) \quad sh = (l, p) \mapsto (\sigma_e, -) \quad \hat{\sigma} = \text{lev}(sp, e_l) \sqcup \text{lev}(sp, e_p) \sqcup \text{lev}(pc) \sqsubset sp(x)
\]

\[
\frac{\{sh, sm, sp, scs, pc\} \xrightarrow{\alpha_{mon}} \{sh, sm, sp[x \mapsto \hat{\sigma} \sqcup \sigma_e]\}, scs, pc\}}
\]

If the label received by the monitor is of type hasField \((x, l, p, e_l, e_p)\), the monitor first searches the security heap for the existence level \( \sigma_e \) of \( p \). The monitor then computes the least upper bound between the level of the program counter, \( \text{lev}(pc) \), the level of \( e_l \), \( \text{lev}(sp, e_l) \), and the level of \( e_p \), \( \text{lev}(sp, e_p) \), assigning the result to \( \hat{\sigma} \). Following the NSU-discipline, the upgrade is only allowed to take place if the \( \hat{\sigma} \) is lower than or equal to the security level of \( x \). Finally, the monitor updates \( x \) to the least upper bound of \( \hat{\sigma} \) with \( \sigma_e \) in the security store.

**Has Field - False:**

\[
o = \text{hasField}(x, l, p, e_l, e_p) \quad (l, p) \notin \text{dom}(sh) \quad sm = sm' \quad l \mapsto (\sigma_s, -) \quad \hat{\sigma} = \text{lev}(sp, e_l) \sqcup \text{lev}(sp, e_p) \sqcup \text{lev}(pc) \sqsubset sp(x)
\]

\[
\frac{\{sh, sm, sp, scs, pc\} \xrightarrow{\alpha_{mon}} \{sh, sm, sp[x \mapsto \hat{\sigma} \sqcup \sigma_s]\}, scs, pc\}}
\]

If the label received by the monitor is of type hasField \((x, l, p, e_l, e_p)\), the monitor searches unsuccessfully for the security levels of \( p \). Then, the monitor searches the security metadata for the structure level \( \sigma_s \) of the location \( l \). Afterwards the monitor computes the least upper bound between the level of the program counter, \( \text{lev}(pc) \), the level of \( e_l \), \( \text{lev}(sp, e_l) \), and the level of \( e_p \), \( \text{lev}(sp, e_p) \), assigning the result to \( \hat{\sigma} \). Following the NSU-discipline, the upgrade is only allowed to take place if the \( \hat{\sigma} \) is lower than or equal to the value of \( x \) in the security store. Finally the security store will update the value of \( x \) to the least upper bound of \( \hat{\sigma} \) with \( \sigma_s \).

**Get Fields:**

\[
o = \text{getFields}(x, l, e_l) \quad sm = l \mapsto (\sigma_s, -) \quad \hat{\sigma} = \text{lev}(sp, e_l) \sqcup \text{lev}(pc) \sqsubset sp(x)
\]

\[
\frac{\{sh, sm, sp, scs, pc\} \xrightarrow{\alpha_{mon}} \{sh, sm, sp[x \mapsto \hat{\sigma} \sqcup \sigma_s]\}, scs, pc\}}
\]

If the label received by the monitor is of type getFields \((x, l, e_l)\), the monitor searches the security metadata for the structure level \( \sigma_s \) associated with the location \( l \). Afterwards, the monitor computes the least upper bound between the level of the program counter, \( \text{lev}(pc) \), and the level of \( e_l \), \( \text{lev}(sp, e_l) \), assigning the result to \( \hat{\sigma} \). Following the NSU-discipline, the upgrade is only allowed to take place if \( \hat{\sigma} \) is lower than or equal to the value of \( x \) in the security store. Finally the security store will update the value of \( x \) to the least upper bound of \( \hat{\sigma} \) with \( \sigma_s \).

**Metadata:**

\[
o = \text{metadata}(x, l, e_l) \quad sm = l \mapsto (\sigma_m, -) \quad \hat{\sigma} = \text{lev}(sp, e_l) \sqcup \text{lev}(pc) \sqsubset sp(x)
\]

\[
\frac{\{sh, sm, sp, scs, pc\} \xrightarrow{\alpha_{mon}} \{sh, sm, sp[x \mapsto \hat{\sigma} \sqcup \sigma_m]\}, scs, pc\}}
\]
If the label received by the monitor is of type metadata \((x, l, e_l)\), the monitor searches the security metadata for the metadata level \(\sigma_m\) associated with \(l\). Afterwards, the monitor computes the least upper bound between the level of the program counter, \(\text{lev}(pc)\), and the level of \(e_l\), \(\text{lev}(sp, e_l)\), assigning the result to \(\hat{\sigma}\). Following the NSU-discipline, the upgrade is only allowed to take place if \(\hat{\sigma}\) is lower than or equal to the value of \(sp(x)\) (the security value of \(x\)). Finally the security store will update the level of \(x\) to the least upper bound of \(\hat{\sigma}\) with \(\sigma_m\).

**Branch:**

\[
o = \langle e, i \rangle \quad \text{lev}(sp, e) = \sigma
\]

\[
\{sh, sm, sp, scs, pc\} \xrightarrow{\sigma}{\text{mon}} \{sh, sm, sp, scs, (\sigma, i) :: pc\}
\]

If the label received by the monitor is of type \(\langle e, i \rangle\), it means that the program is branching on the value of \(e\) at the command with index \(i\); hence the monitor extends the current pc with the pair \((\sigma, i)\) consisting of the level of \(e\) at index \(i\).

**Merge:**

\[
o = \triangleright (\bar{i}) \quad pc_{\bar{i}} = pc'
\]

\[
\{sh, sm, sp, scs, pc\} \xrightarrow{\bar{i}}{\text{mon}} \{sh, sm, sp, scs, pc'\}
\]

If the label received by the monitor is of type \(\triangleright (\bar{i})\), it means that the current program point corresponds to a merging point of all the conditional gotos with indexes in \(\bar{i}\). Accordingly, the monitor removes all the consecutive top entries in the pc associated with gotos with indexes \(\bar{i}\). This is done through the projection that can be seen in the table below.

**Merge Rules**

\[
[ ] \downarrow \bar{i} = [ ]
\]

\[
((\sigma, j) : pc') \downarrow \bar{i} = pc' \downarrow \bar{i}, \ j \in \bar{i}
\]

\[
((\sigma, j) : pc') \downarrow \bar{i} = (\sigma, j) : pc', \ j \notin \bar{i}
\]

### 4.3.3 Example

In order to illustrate how our monitor works let us examine two monitored executions of the program given in Figure 4.6. Figure 4.7 represents the execution of a JSIL program together with the labels that are sent to the information flow monitor. In order to illustrate the behaviour of the monitor, we have designed two different starting scenarios, that differ on the security level of the variable \(x\).

**Low \(x\):** In this case, illustrated in Figure 4.8, the program starts by setting the variable \(w\) to a boolean indicating whether or not the object \(l_o\) has the property "get". Since \(l_o\) has high structure security level, the monitor upgrades the level of \(w\) to \(H\) as stated in the Rule HAS FIELD - FALSE of the monitor semantics. Since the execution branches on the value of \(w\), the monitor extends the current program counter with the pair \((H, 2)\), meaning that the remaining commands will get executed in a High context. Consequently, the monitored execution of the PROPERTY LOOKUP in line 3 of the program gives rise to
an information flow exception, as the program is trying to reset the value of the low variable \( x \) within a high context.

**High \( x \):** In this case, illustrated in Figure 4.9, the program starts by setting the variable \( w \) to a boolean indicating whether or not the object \( l_o \) has the property "get". Since \( l_o \) has high structure security level, the monitor upgrades the level of \( w \) to \( H \) as stated in the Rule HAS FIELD - FALSE of the monitor semantics. Since the execution branches on the value of \( w \), the monitor extends the current program counter with the pair \((H, 2)\), meaning that the remaining commands will get executed in a High context. Unlike in the previous example, the level of \( x \) is \( H \) meaning it will pass the constraint of the PROPERTY LOOKUP rule, and effectively assigning \( l_p \) to \( x \). Now, the execution branches on \( x \), the monitor extends the current program counter with the pair \((H, 4)\). Then, the program checks again whether or not the object \( l_o \) has the property "get", and assigns the boolean to the variable \( w \). The program branches in the value of \( w \) and extend the program counter with \((H, 2)\). Finally the program pops all the top entries of the pc with indexes 2 or 4, restoring a low execution context.

### 4.4 Running Example

In this section, we illustrate how the monitored JSIL semantics works by appealing to the JS running example given in Section 1.2. Figure 4.10 shows a stylised JSIL compilation of the body of the function `getRightBit1`. This snippet of JSIL code still has the same information flow leak as the corresponding JS program, given in Figure 1.4. Recall that the mentioned JS program has an information flow leak via the length of the array \( a \), which has low security level. More concretely, depending on the parity of the high variable \( x \), the program will set the length of the array \( a \) to either 1 or 2.

In order to understand how this program is executed by the information flow monitor, we present two execution traces which explain the cases where where the input \( x \) of a program is even (Figure 4.14) and not even (Figure 4.12). The traces detail the changes in the program counter, security store, security heap and security metadata. Below we give a high-level description of the figures.

**Input is not even:** In Figure 4.12, the program starts by creating the array \( a \) and by populating it (lines 0-3). Afterwards it checks whether or not \( x \) is even (line 4). As the value of \( x \) is not even, the control jumps to line 8, updating the program counter level. Now the program merges the branches associated with index 4 (popping the top tuples of the program counter with index 4). Then, the program creates a reference \( aRef \) to the property \( length \) of the array \( a \) (line 9), which has low security level, and uses it to set the \( length \) of the array to 1 (line 10). Now, the program checks the value of the current length of the array, assigning it to the new variable \( y \) (line 11), which has low security level as the current context level
Figure 4.7: Example of running a program - instrumentalized
Figure 4.8: Example of running a program - Monitor
Figure 4.9: Example of running a program - Monitor
is also low. Finally, depending on the value of $y$ the program returns (line 12-16). The return value has low security level as the security level of $ret$ is low.

0     a:=initArray()
1     putValue(a,'0','a')
2     putValue(a,'1','b')
3     putValue(a,'2','c')
4     goto[(x%2=0)] then1 next1
5     then1 desc := vDesc('b', false, true, true)
6     defineOwnProperty(a, '1',desc)
7     goto next1
8     next1 merge 4
9     aRef:=oRef(a, 'length')
10    putValue(aRef, 1)
11    y:= getValue(aRef)
12    goto[y=1] then2 else2
13    then2 ret:=1
14    return
15    else2 ret:=0
16    return

Figure 4.10: Running Example in JSIL

**Input is even:** In Figure 4.13, the program starts by creating the array $a$ and by populating it (lines 0-3). Afterwards it checks whether or not $x$ is even (line 4), updating the level of the program counter to be high and jumping to line 5. Then it creates a descriptor $desc$ (line 5), which has High security level as it is created within a high context. When the program tries to set the property $1$ of the array $a$ to $desc$ (line 6), the monitor throws an error as this constitutes a no sensitive upgrade, as we are assigning a low property in a high context. To prevent this NSU we simply need to upgrade the level of the property $1$ and the $length$ of the array before entering a high context. We do this using the upgrade command. The modified code is shows in Figure 4.11 and the next paragraph explains its monitored execution.

**Input is even with upgrade commands:** Figure 4.14, shows the execution trace of the information flow monitor when running the code snippet given in Figure 4.11, the JSIL running example with the appropriate upgrades. The program starts by creating the array $a$, by populating it (lines 0-3), and by upgrading the security level of both the property "length" and the property "1" of $a$ (line 4-5). Afterwards, it checks whether or not $x$ is even (line 6), updating the level of the program counter to be high and jumping to line 7. Then, it creates a descriptor $desc$ that cannot be deleted (line 7). Now, the program sets the property of position "1" of $a$ to be $desc$ (line 8). In this case, the command is executed successfully because of the upgrades performed in lines 4-5. Then the program jumps to $next1$ (line 9). Afterwards, the program merges the branches associated with index 6 (popping the top levels of the program counter with index 6). Then, the program creates a reference $aRef$ to the property $length$ of
the array a (line 11), which has high security level, and uses it to set the length of the array to 1 (line 12).
Now, the program checks the value of the current length of the array, assigning it to the new variable y
(line 13), which has a high security level as the level of aRef is also high. Finally, depending on the value
of y the program returns 0 or 1 (line 14-18). The return value has high security level as the security level
of ret is also high.

```
a:=initArray()
putValue(a,'0','a')
putValue(a,'1','b')
putValue(a,'2','c')
upgProp(a,'1',H)
upgProp(a,'length',H)
goto[(x%2=0)] then1 next1
then1 desc := vDesc('b', false, true, true)
defineOwnProperty(a, '1',desc)
goto next1
next1: aRef:= merge 6
aRef:=oRef(a, 'length')
putValue(aRef, 1)
y:= getValue(aRef)
goto[y=1] then2 else2
then2: ret:=1
return
else2 ret:=0
return
```

Figure 4.11: Running Example with upgrades JSIL
Figure 4.12: Running example when x is odd
Figure 4.13: Running example when x is even
Figure 4.14: Running example when x is even, now with upgrades
Chapter 5

Implementation

In this chapter we explain the Rosette implementation of the JSIL monitor described in Chapter 4. We first define the data structures used to model in Rosette the JSIL security domains (Section 5.1). Afterwards, we explain the implementation of the monitor semantics, introduced in 4, into code (Section 5.2). Finally, describe our solution for dealing with the label creep problem, resulting from the fact that JSIL in an unstructured goto language (Section 5.3).

5.1 Data Structures

The table below presents the Rosette implementation of the JSIL security domains. Rosette is a subset of Racket with very limited support for complex data structures. In order to overcome this hurdle we have developed our structures only using lists, which are fully covered by Rosette. The security heap is modelled as a list of pairs, where the first element of each pair is a location and the second one is the security labelling of the object stored at that location. The security labelling of an object is a list of pairs of the form \((p(l_1, l_2))\), where \(p\) is a property and \((l_1, l_2)\) are its corresponding existence and value levels. The security store is modelled as a list of pairs of the form \((x, l)\), where \(x\) is a variable and \(l\) is the corresponding value level. The security metadata is modelled as a list of pairs of the form \((loc(l_1, l_2))\), where \(loc\) is a location and \((l_1, l_2)\) correspond to its structure level and metadata level. The program counter is modelled as a list of pairs of the form \((l, i)\), where \(l\) is a security level and \(i\) is the index of the command that branched. Finally the security call stack is a list modelled as \((x(pc, sp))\), where: \(x\) is the return index for the current call, \(pc\) is the program counter at the time of the call, \(sp\) is the security store at the time of the call.
5.2 Rule Implementations

We have implemented the monitor as a racket function that pattern matches on the information flow event generated by the JSIL semantics. Each of the pattern-matching case corresponds to a information flow event (see figure 3.5). The information flow events are in one to one correspondence with information flow monitor rules, hence each pattern matching case corresponds to an information flow monitor rule, and its implementation precisely follows the meta theoretical formulation presented in Chapter 4.

In order to explain the implementation of the rules in a more intuitive way, we pair up each rule with its corresponding rosette implementation. Using a color scheme to connect every line of code to its corresponding mathematical counterpart. Accordingly we use: blue for the labels received by the monitor, grey for the lines that fetch relevant security levels, green for computations on security levels, for instance the computation of the context level, red for the security constraints, and finally violet for security level upgrades. The implementation is given in Figures 5.1 to 5.4.

As can be seen in Figure 5.1, through the color code we can easily associate to every line of code its corresponding. The rosette code follows the rule’s semantics very closely, which have been thoroughly explained in chapter 4, therefore there is no need to explain the code any further in this section.

5.3 JS information flow monitor

Recall that we obtain an information flow monitor for JS by first compiling the given javascript program to JSIL and then applying our JSIL information flow monitor. This methodology presents two problems:

(1) how to label JS resources with information flow levels?

(2) how to cope with the label creep [56] problem caused by JSIL unstructured control flow?
\[
\begin{align*}
\text{PROPERTY ASSIGNMENT-1} & \\
\{sh, sm, sp, scs, pc\} & \xrightarrow{\text{mon}} \{sh, sm, sp, scs, pc\} \\
\text{PROPERTY ASSIGNMENT-2} & \\
\{sh, sm, sp, scs, pc\} & \xrightarrow{\text{mon}} \{sh, sm, sp, scs, pc\} \\
\text{OBJECT CREATION} & \\
\{sh, sm, sp, scs, pc\} & \xrightarrow{\text{mon}} \{sh, sm, sp, scs, pc\} \\
\text{ASSIGNMENT} & \\
\{sh, sm, sp, scs, pc\} & \xrightarrow{\text{mon}} \{sh, sm, sp, scs, pc\} \\
\text{HAS FIELD - TRUE} & \\
\{sh, sm, sp, scs, pc\} & \xrightarrow{\text{mon}} \{sh, sm, sp, scs, pc\} \\
\end{align*}
\]

Figure 5.1: Implementation part 1
HAS FIELD - FALSE

\[
\begin{align*}
o &= \text{hasField}(x, l, e, \varphi, \epsilon, \nu) \\
(l, p) \notin \text{dom}(sh) & \implies \text{sm} = \text{sm}' \equiv l \mapsto (\sigma, -) \mapsto - \\
\sigma &= \text{lev}(sp, e) = \text{lev}(sp, \epsilon) = \text{lev}(pc) \\
\{sh, sm, sp, scs, pc\} \xrightarrow{\text{mon}} \{sh, sm, sp, scs, pc\}
\end{align*}
\]

PROPERTY DELETION

\[
\begin{align*}
o &= \text{delete}(l, p, e, \varphi, \epsilon, \nu) \\
\text{sm} &= l \mapsto (\sigma, -) \mapsto - \\
\sigma &= \text{lev}(sp, e) = \text{lev}(sp, \epsilon) = \text{lev}(pc) \\
\{sh, sm, sp, scs, pc\} \xrightarrow{\text{mon}} \{sh_1, sm, sp, scs, pc\}
\end{align*}
\]

CALL

\[
\begin{align*}
o &= \text{call}(e, x, \varphi, \epsilon, \nu) \\
\text{lev}(sp, e) &= \sigma_1 \\
\sigma_1 &= \text{lev}(sp, \epsilon) = \text{lev}(pc) \\
\{sh, sm, sp, scs, pc\} \xrightarrow{\text{mon}} \{sh, sm, sp, scs, pc\}
\end{align*}
\]

MERGE

\[
\begin{align*}
o &= \text{merge}(n) \\
\text{pc} &= \text{pc}' + \text{length}(\text{pc}') = n \\
\{sh, sm, sp, scs, pc\} \xrightarrow{\text{mon}} \{sh, sm, sp, scs, pc\}'
\end{align*}
\]

METADATA

\[
\begin{align*}
o &= \text{metadata}(x, l, e, \varphi, \epsilon, \nu) \\
\text{sm} &= l \mapsto (\sigma, -) \\
\sigma &= \text{lev}(sp, e) = \text{lev}(pc) \\
\{sh, sm, sp, scs, pc\} \xrightarrow{\text{mon}} \{sh, sm, sp, scs, pc\}
\end{align*}
\]

Figure 5.2: Implementation part2
**BRANCH**

\[
o = \lambda (e) \quad lev(sp, e) = \sigma
\]

\[
\{sh, sm, sp, scs, pc\} \xrightarrow{mon} \{sh, sm, sp, scs, pc\}
\]

\[
\text{let}^* (\text{sigma} (\text{expr-lev} (\text{sstore-sel sconf}) e))
\]

\[
\{pc-2 (\text{add-pc} (\text{pc-sel sconf}) \text{sigma index}))\}
\]

\[
\text{updt-pc sconf pc-2}\]

**GET FIELDS**

\[
o = \text{getFields} \{x, l, e\}
\]

\[
\text{let}^* ((\text{sigma-s} (\text{struct-lev} (\text{meta-sel sconf}) l))
\]

\[
\{sh, sm, sp, scs, pc\} \xrightarrow{mon} \{sh, sm, sp1, scs, pc\}
\]

**LOOKUP**

\[
o = \text{p-lookup} \{x, l, p, e\}
\]

\[
\text{let}^* ((\text{sigma-v} (\text{val-lev} (\text{heap-lookup mheap} l p)))
\]

\[
\{sh, sm, sp, scs, pc\} \xrightarrow{mon} \{sh, sm, sp1, scs, pc\}
\]

\[
\text{upd-store sconf store-1})
\]

\[
\text{raise (formatConstraint o}
\]

\[
\text{ctx-lev}
\]

\[
\text{var-lev (sstore-sel sconf) x})))))
\]

**UPGRADE VARIABLE**

\[
o = \text{upgVar} \{x, \sigma\}
\]

\[
\text{let}^* ((\text{sigma} (\text{expr-lev} (\text{sstore-sel sconf}) e)\]

\[
\{sh, sm, sp, scs, pc\} \xrightarrow{mon} \{sh, sm, sp', scs, pc\}
\]

\[
\text{upd-store sconf store-1})
\]

\[
\text{raise (formatConstraint o}
\]

\[
\text{pc-lev (pc-sel sconf) x new-lev})\]

\[
\text{store-1 (updt-store-var (astore-sel sconf) x new-lev))})
\]

\[
\text{upd-store sconf store-1})
\]

\[
\text{raise (formatConstraint o}
\]

\[
\text{pc-lev (pc-sel sconf) x new-lev})\]

\[
\text{store-1 (updt-store-var (astore-sel sconf) x new-lev))})
\]

**Figure 5.3: Implementation part 3**
Figure 5.4: Implementation part 4
5.3.1 labelling JavaScript resources

In order to specify the security levels of JavaScript variables, objects and object properties, we have extended the syntax of JavaScript with dedicated upgrade commands analogous to those proposed for JSIL. The JavaScript upgrade commands are:

- `upgVar(x, σ)` for upgrading the level of variable `x` to `σ`;
- `upgProp(o, p, σ)` for upgrading the value level of property `p` of object `o` to `σ`;
- `upgPropE(o, p, σ)` for upgrading the value level of property `p` of object `o` to `σ`;
- `upgStruct(o, σ)` for upgrading the structure level of the object `o`;
- `upgMeta(o, σ)` for upgrading the metadata level of the object `o`.

Hence, in order to label a given JavaScript resource with a given security level the JavaScript programmer simply has to use the appropriate upgrade command. For instance, we write `upgVar(x, "H")` to mean that the JavaScript variable `x` has security level High. As in JSIL unlabelled resources are assumed to be public.

In order to take the JS upgrade commands into account during the information flow analysis, we have extended the JS-2-JSIL [1] compiler with support for these commands.

5.3.2 Label creep and unstructured control flow

Every time a JSIL program branches when executing a conditional goto command, the current program counter is extended with the security level of the guard of the goto. The only way to pop this level out of the program counter is to place a merge command at the merging point of both branches of the goto. This is not done automatically. Hence, if the programmer fails to place merging commands in the appropriate merging points, the program counter will continuously increase, resulting in the so called label creep problem [56]. In a nutshell, after branching on a single high guard, all the following commands will get executed in a high context, potentially triggering false no sensitive upgrade exceptions.

The compilation of a JavaScript program into JSIL does not place merging commands at the merging points of conditional goto branches. Due to the size of the generated JSIL programs (approximately 10 times the size of the original JS program), it is unreasonable to expect the user to place these merge commands manually. To counter this problem we extend the JS-2-JSIL compiler with a post-processing phase that analyses the generated JSIL program and places the merge commands in the appropriate merging points. We divide the post-processing phase into the following steps:

- Construct the control flow graph of every procedure in the given JSIL program;
- Compute the post-dominators tree of each control flow graph;
- Extend each control flow graph with new nodes representing the merging commands;
- Convert the newly generated control flow graphs into a JSIL program.
Even though the JS-2-JSII compiler is implemented in Ocaml, we have chosen to implement all the post-processing steps in racket. This choice facilitated the integration of the post-processing phase with our implementation of the information flow analysis. Below, we explain each step, appealing to the example of the JSIL program given in Figure 5.5.

**Program to control flow graph:** In this step we construct the control flow graph of every procedure in the JSIL program given as input. To this end, we make use the Racket Generic Graph Library [57].

A JSIL program consists of a list of JSIL procedures. Therefore, we create a list of control flow graphs, one for every procedure in the program. Each node of these newly created graphs represents a command of the original program. Every node is labelled with the index of its corresponding command, and has outgoing edges to all of the possible successors of the command. Given the command goto [h] then1 merge1 at index 4, where then1 and merge1 are the labels of the commands 5 and 7, the node 4 has two outgoing edges linking it to nodes 5 and 7. Analogously, given the command h := false at index 3 The return and error commands do not have outgoing edges. Figure 5.8 shows the graph of the program given in Figure 5.5.

**Post-dominator tree:** In this step we compute the post-dominator tree [58] of each control flow graph of the given JSIL program. Intuitively, a node \( z \) is said to post-dominate a node \( n \) if all paths to the exit node of the graph starting at \( n \) must go through \( z \). Furthermore, we say that \( n \) is an immediate post-dominator [58] of \( z \) if: (1) \( n \) is a post-dominator of \( z \); (2) \( n \) is different from \( z \); and the (3) if a node \( m \neq z \) post-dominates \( z \), then \( m \) also post-dominates \( n \). For instance, in the graph given in Figure 5.8, node 7 immediately post-dominates nodes 4 and 6, because all the paths from these nodes to the exit go through node 7.

The post-dominator tree connects each node to all of its immediate post-dominators. The post-dominator tree of the graph given in Figure 5.8 is shown in Figure 5.9. This tree tells us precisely which are the merging points of the given control flow graph. In a nutshell, the merging points coincide with the nodes that immediately post-dominate more than one node. For instance, the merging points of the graph in Figure 5.8 are the nodes 7 and 12, which is immediately apparent by inspection of the post-dominator tree, Figure 5.9.

Instead of computing the post-dominator tree directly, we compute the dominator tree of the transposed graph. Dominators and immediate dominators are the dual concepts of post-dominators and immediate post-dominators. While post-dominators refer to the paths from the given node to the exit node, dominators refer to the paths from the entrance node to the given node. Intuitively, the post-dominator tree of a given graph coincides with the dominator tree of the transposed graph.

In order to compute the dominator tree of the transposed graph, we first use a standard worklist algorithm [59] for computing the dominators of every node in the graph, and then use a simple DFS algorithm [60] to obtain the immediate post-dominator tree.

**Inserting merge commands:** In this step we extend the control flow graph with new nodes representing the merge commands. Given the post-dominator tree, we know exactly where to put the merge commands: merge commands need to be placed immediately before each node that post-dominates more than one node. Note that we have to place the merge command before the merging point command,
as this command can be a goto command, effectively preventing the merge command from getting executed. Each merge command has the list of nodes that are immediately post-dominated by the merging point command as its argument. For instance, in Figure 5.10 we can see the change in the control flow graph of adding the merge commands \texttt{merge}(6, 4) in node 7, which has seen previously is a merging point. Figure 5.10 shows the result of adding the merges to the control flow graph given in Figure 5.8.

**Recovering the program:** In this step, we rewrite the extended control flow graph as a JSIL program, obtaining a version of the original program with the appropriate merge commands. Figure 5.6 gives us the final updated program in JSIL.

In racket we represent JSIL programs as s-expressions [61]. Hence, the final output of the post-processing phase is immediately generated as an s-expression. In Figure 5.7, we show the output of the post-processing phase when applied to the JSIL program given in Figure 5.5.
#lang racket
(require (file "mem_model.rkt"))
(require (file "interpreter_iflow.rkt"))
(define prog
  (program
    (procedure
      "main"
      (args)
      (body
        (new o null)
        (h-assign o "p" #t)
        (v-assign l #t)
        (goto h 5 8)
        (skip)
        (h-assign o "p" #f)
        (merge (6 4))
        (skip)
        (h-read laux o "p")
        (goto laux 11 14)
        (skip)
        (v-assign l #f)
        (merge (10 12))
        (skip)
        (assert (and (= h #f) (= l #f)))
        (success)
        (v-assign xret #t))
    (ret-ctx 'xret 17)
    ()
  )))
(run-program prog (make-heap))

Figure 5.7: Updated program in Racket
Figure 5.8: Example Graph

Figure 5.9: Example post dominators
Chapter 6

Evaluation

In this section we discuss the evaluation process we applied to our implementation of the information flow monitor. Firstly, we give a brief explanation on how to use the tool. Then, we describe the JSIL and JavaScript test suites that we have designed to evaluate the information flow monitor.

6.1 How to use

The proposed information flow monitor does not run on its own. It is designed to be a component of an analysis pipeline that also includes the JS-2-JSIL compiler and the post-processing phase. Hence, in order to apply the monitor to a given JS program, the user needs to first compile the given program to JSIL, then apply the post-processing phase, and finally run the monitored semantics.

When executing JSIL programs generated by the JS-2-JSIL compiler, the JSIL interpreter expects a number of auxiliary files to be present in the current working directory. To this end, we make use of a dedicated script, called setup_environment.sh, which creates such folder.

Below, we detail the steps required for running the information flow monitor on a given JavaScript program:

1. Create the environment folder:

   
   $ ./setup_environment.sh

2. Compile the input program to JSIL in the environment folder:

   
   $ ./js2jsil.native -file <JS-FILE>
   e.g. $ ./js2jsil.native -file HelloWorld.js

3. Serialize the JSIL program as a Racket s-expression:

   
   $ ./jsil2rkt.native -file <JSIL-FILE> -js -graph
   e.g. $ ./jsil2rkt.native -file HelloWorld.jsil -js -graph
This step generates a Racket program that will in turn generate the Racket representation of the given JSIL program, extending it with merge commands. For instance, when running JSIL-2-RKT on a given JSIL program, say *HelloWorld.jsil*, we first obtain a program *HelloWorld.rkt*. Note that this program is not a Racket representation of the given JSIL program, but rather a racket program that applies the post-processing phase to the input JSIL program, generating the final Racket representation of the input program with merge commands.

4. Run the output of the previous step in order to obtain the final Racket representation of the JSIL program:

```
$racket <RACKET-FILE-1>
e.g. $racket HelloWorld1.rkt
```

This step generates a Racket program that will call the information flow monitor on the given JSIL program extended with merge commands. In this example the program *HelloWorld1* generates the program *HelloWorld2*, which has the merge commands.

5. Run the final Racket program:

```
$racket <RACKET-FILE-2>
e.g. $racket HelloWorld2.rkt
```

Information flow exceptions thrown during the monitored execution of the program are directly presented to the user.

In order to run the monitored semantics directly on a JSIL program we proceed as we do for JavaScript programs, ignoring the first compilation step and the flag -js in the second step.

**Scripts**: In order to facilitate the process mentioned above we have developed scripts that automatize the previous steps. To do so our scripts are run in the environment folder. Then, the script finds the appropriate test folder for the test suite we want to run, and follows the aforementioned steps to run the program. Figure 6.1 shows how our script that runs JSIL programs goes through the folders to do the post processing steps. Finally, after the post-processing the script runs the final racket program, recording the execution time.

### 6.2 Tests

In order to test our monitor, we have designed a JavaScript and a JSIL test suites. Each test suite includes positive tests and negative test targeting each type of information flow leak (see Table 4.2).

We perform the entire evaluation on a machine with an Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz, DDR3 Synchronous 1600 MHz RAM 6GB, and a 750GB hard-drive running kali with release 2019.2.
Figure 6.1: Post-processing phase example
We have designed a JSIL test suite for our information flow monitor containing a number of positive and negative unit tests per type of information flow leak. While negative tests are expected to throw an information flow exception, positive tests are expected to run normally. In order to run the JSIL test suite we have ran the script

$./test_iflow_JSIL_graph.sh

Results are shown in Table 6.1. For each type of leak we show: (1) the number of positive tests (# Tests (+)); (2) the number of negative tests (# Tests (-)); (3) the average number of lines per test # Lines (avg); (4) the sum of the post-processing time in seconds (# PP Time); (5) the sum of the runtime with a monitor (Runtime-mon); and (6) the sum of the runtime when no monitor is used (Runtime-mon).

As shown in Table 6.1 the difference between running the program with a monitor or without one in our small testsuite is minimal. Hence, we believe this monitor may be a good subject to work upon in the further future.

**Designing information flow tests:** In order to understand how we have designed each test in the JSIL testsuite, let us consider the following example depicting a leak of Type 1:

<table>
<thead>
<tr>
<th>Type of leak</th>
<th># Tests (+)</th>
<th># Tests (-)</th>
<th># Lines (avg)</th>
<th># PP Time</th>
<th>Runtime-mon</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
<td>6</td>
<td>20.29</td>
<td>19.603</td>
<td>32.761</td>
<td>27.699</td>
</tr>
<tr>
<td>II</td>
<td>1</td>
<td>1</td>
<td>26.5</td>
<td>6.849</td>
<td>9.145</td>
<td>7.892</td>
</tr>
<tr>
<td>III</td>
<td>1</td>
<td>1</td>
<td>26.5</td>
<td>5.652</td>
<td>9.048</td>
<td>7.925</td>
</tr>
<tr>
<td>IV</td>
<td>1</td>
<td>1</td>
<td>27.5</td>
<td>5.587</td>
<td>9.111</td>
<td>7.924</td>
</tr>
<tr>
<td>V</td>
<td>1</td>
<td>1</td>
<td>28.5</td>
<td>5.574</td>
<td>8.839</td>
<td>7.907</td>
</tr>
<tr>
<td>VI</td>
<td>1</td>
<td>1</td>
<td>29.5</td>
<td>6.342</td>
<td>8.932</td>
<td>7.887</td>
</tr>
<tr>
<td>VII</td>
<td>1</td>
<td>1</td>
<td>29.5</td>
<td>6.038</td>
<td>9.827</td>
<td>7.898</td>
</tr>
</tbody>
</table>

Table 6.1: JSIL test suite
The JSIL program in the left is a positive test, as we can see there is no information flow leak as all the variables are low. Meanwhile the JSIL program to the right is a negative test, as there is a high goto effectively increasing the context level of the program, creating a information flow leak as the program tries to assign a low variable in a high context. We have used this methodology to create the JSIL test suite.

6.2.2 JavaScript

We have designed a JavaScript test suite for our information flow monitor containing a number of positive and negative unit tests per type of information flow leak. While negative tests are expected to throw an information flow exception, positive tests are expected to run normally. In order to run the JavaScript test suite we have ran the script

```
$ ./test_iflow_JS_graph.sh
```

Results are shown in Table 6.2. For each type of leak we show: (1) the number of positive tests (# Tests (+)); (2) the number of negative tests(# Tests (-)); (3) the average number of JavaScript lines per test # JS Lines (avg); (4) the average number JSIL of lines per test # JSIL Lines (avg); (5) the sum of the post-processing time in seconds (# PP Time); (6) the sum of the runtime with a monitor (Runtime-mon); and (7) the sum of the runtime when no monitor is used (Runtime).

Table 6.2 shows the difference between running the program with a monitor or without one in our limited testsuite is minimal, although if we consider the post-precessing time the price becomes much higher. Hence, we believe this monitor may be a good subject to work upon.
Table 6.2: JavaScript test suite

Designing information flow tests: In order to understand how we have designed each test in the JavaScript test suite, let us consider the following example depicting a leak of Type 1:

**TYPE: 1**

without upgrade

```javascript
var h = false;
var l = true;
var laux = true;
if (h) {
    laux = false;
}
if (laux) {
    l = false;
}
```

with upgrade

```javascript
var h = false;
upgVar(h, "H");
var l = true;
var laux = true;
if (h) {
    laux = false;
}
if (laux) {
    l = false;
}
```

The JavaScript program in the left is a positive test, as we can see there is no information flow leak as all the variables are low, i.e. there is no high information to leak. Meanwhile the JavaScript program to the right is a negative test, as there is an assignment to a low level variable in a high context, therefore the monitor throws an information flow exception. We have used this methodology, of creating a positive and negative test for each type of leak, to create the JavaScript test suite.
Chapter 7

Conclusions

JavaScript is difficult to analyse, mostly due to being a very dynamic language and the heterogeneity of browsers in use. This thesis presents a static monitor that prevents information flow leaks in JavaScript. This analysis is simplified by using the intermediate language JSIL, a goto language with an existing symbolic execution tool. Our monitor analyses the information flow present in the JSIL code execution, throwing an error every time a program would leak information, even if it is in a recondite way. These JSIL information flow leaks correspond to information flow leaks in the original JavaScript, as the compiled JSIL programs are semantically equivalent to the original programs. In this final section we show what was done in order to get the information flow monitor to work (Section 7.1) and what was left to be done in future work (Section 7.2).

7.1 Achievements

The development of the information flow monitor included the following technical contributions:

- Creation of new commands for the JSIL language in order to support a monitor, these commands allow the manipulation of security resources;
- Creation of a general monitored semantics for the JSIL language, this creates an abstraction between the JSIL code itself and the monitor, essentially making it much easier to change monitors;
- Formally define the syntax and semantics for an information flow monitor that analyses JavaScript through the JSIL language. This monitor follows the no sensitive upgrade discipline;
- Implement the information flow monitor in racket;
- Deal with the label creep problem that occurred within our monitor. To do this we had to analyse the program in order to know where to insert merges to reduce the context level of the execution.
- Test our monitor in racket, through a battery of unit tests.

With this, we now have a tool that is capable of catching information flow leaks in JavaScript, filling a void in the state of the art, as there were no such tools before, to the best of our knowledge.
7.2 Future Work

Even though this solution is implemented and functional, further work can give stronger guarantees, such as:

• The application of this tool to a real program, e.g. the Tagus voting tool, which is being developed in IST as a voting solution suitable for Universities and other similar organisations;

• Prove the soundness of the monitor rules. In order to ensure our rules are sound and correct, they need to be tested.

• Perform symbolic execution tests, these will tests allow us to automatize the evaluation process, as these tests would go through all the possible execution states of the program.

• Apply the JSIL general monitored semantics to different kinds of monitors, i.e. monitors that do not enforce information flow security. As our general monitored semantics for the JSIL language separates the monitor from the language itself we can monitor other properties instead of focusing on information flow.

With this additional work we believe that the monitor will be useful and achieve appreciable impact. This project creates a new form of JavaScript analysis when compared to the state of the art.
Bibliography


[57] Graph racket library. [https://docs.racket-lang.org/graph/index.html](https://docs.racket-lang.org/graph/index.html).


Appendix A

JSIL Symbolic Monitor

In this chapter we will analyse symbolic JSIL. Starting by it going through the symbolic semantics of the JSIL language (Section A.1). Then we will propose a symbolic monitor and analyse how it differs from the monitor presented in Chapter 4.

A.1 JSIL Symbolic Semantics

The Table below we defines the symbolic domains for the JSIL language. A symbolic heap maps to each pair of symbolic locations and symbolic properties a symbolic value. The symbolic metadata maps to each symbolic location a symbolic value. Finally a symbolic store maps for each concrete variable a symbolic value.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Symbolic Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>$\hat{h} \in \hat{H}_{\text{Heap}} \times \text{Loc} \times \text{Prop} \rightarrow \text{Val}$</td>
</tr>
<tr>
<td>Metadata</td>
<td>$\hat{m} \in \hat{M}_{\text{Metadata}} \times \text{Loc} \rightarrow \text{Val}$</td>
</tr>
<tr>
<td>Store</td>
<td>$\hat{\rho} \in \hat{S}_{\text{Store}} \times \text{Var} \rightarrow \text{Val}$</td>
</tr>
</tbody>
</table>

Table A.1: JSIL Symbolic Semantics

A.1.1 Basic Command

The semantics of symbolic JSIL basic commands is given in Figure A.1 in a small-step style; semantic transitions have the form $\{\hat{h}, \hat{m}, \hat{\rho}, \pi, \text{bcmd}\} \xrightarrow{o} \{\hat{h}, \hat{m}, \hat{\rho}\}$, where:

- $\{\hat{h}, \hat{m}, \hat{\rho}, \pi, \text{bcmd}\}$ is the initial configuration consisting of a symbolic heap, a symbolic metadata table, a symbolic store, a path condition (which is a formula that accumulates constraints on the symbolic variables that guide the execution to the current symbolic state) and a basic command to execute;

- $o$ is an event label to be given to the information monitor;

- $\{\hat{h}, \hat{m}, \hat{\rho}\}$ is the final configuration consisting of the final symbolic heap, symbolic metadata and symbolic store.
Skip
\{m, p, \pi, \text{skip}\} \xrightarrow{\text{cmd}} \{m, p\}

**OBJECT CREATION**
\[
[e_1]_\rho = \hat{v}\text{ if } \rho \notin \text{dom}(\hat{m})
\]
\[
\begin{align*}
\rho &= \text{new}(x, e) \\
\{m, p, \pi, x := \text{new}(e)\} &\xrightarrow{\text{cmd}} \{m', p[x \rightarrow \hat{v}]\}
\end{align*}
\]

**P-ASSIGN 1**
\[
[e_1]_\rho = \hat{l} \\
[e_2]_\rho = \hat{p}
\]
\[
\begin{align*}
\rho &= \text{p-assign}(\hat{l}, \rho, e_1, e_2) \\
\{m, p, \pi, x := [e_1, e_2]\} &\xrightarrow{\text{cmd}} \{m', \hat{m}, \hat{p}\}
\end{align*}
\]

**DELETE**
\[
[e_1]_\rho = \hat{l} \\
[e_2]_\rho = \hat{p}
\]
\[
\begin{align*}
\rho &= \text{delete}(\hat{l}, \rho, e_1, e_2) \\
\{m, p, \pi, x := \text{hasField}(e_1, e_2)\} &\xrightarrow{\text{cmd}} \{m', \hat{m}, \hat{p}\}
\end{align*}
\]

**HASFIELD-FALSE**
\[
\begin{align*}
[e_1]_\rho = \hat{l} \\
[e_2]_\rho = \hat{p}
\end{align*}
\]
\[
\begin{align*}
\rho &= \text{hasField}(\hat{l}, \hat{p}, e_1, e_2) \\
\{m, p, \pi, x := \text{hasField}(e_1, e_2)\} &\xrightarrow{\text{cmd}} \{m', \hat{m}, \hat{p}[x \rightarrow \text{false}]\}
\end{align*}
\]

**GETFIELDS**
\[
[e_1]_\rho = \hat{l} \\
[e_2]_\rho = \hat{p}
\]
\[
\begin{align*}
\rho &= \text{getFields}(\hat{l}, \hat{p}, e_1, e_2) \\
\{m, p, \pi, x := \text{getFields}(e)\} &\xrightarrow{\text{cmd}} \{m, p, [x \rightarrow \rho_1, \rho_2, \ldots]\}
\end{align*}
\]

**ASSIGN**
\[
[e_1]_\rho = \hat{v}
\]
\[
\begin{align*}
\rho &= \text{assign}(x, e) \\
\{m, p, \pi, x := e\} &\xrightarrow{\text{cmd}} \{m, p[x \rightarrow \hat{v}]\}
\end{align*}
\]

**LOOKUP**
\[
[e_1]_\rho = \hat{l} \\
[e_2]_\rho = \hat{p}
\]
\[
\begin{align*}
\rho &= \text{p-lookup}(\hat{l}, \rho, \hat{p}, e_1, e_2) \\
\{m, p, \pi, x := [e_1, e_2]\} &\xrightarrow{\text{cmd}} \{m', \hat{m}, \hat{p}[x \rightarrow \hat{v}]\}
\end{align*}
\]

**P-ASSIGN 2**
\[
[e_1]_\rho = \hat{l} \\
[e_2]_\rho = \hat{p}
\]
\[
\begin{align*}
\rho &= \text{p-assign}(\hat{l}, \rho, e_1, e_2) \\
\{m, p, \pi, x := [e_1, e_2]\} &\xrightarrow{\text{cmd}} \{m', \hat{m}, \hat{p}[x \rightarrow \hat{v}]\}
\end{align*}
\]

**HASFIELD-TRUE**
\[
[e_1]_\rho = \hat{l} \\
[e_2]_\rho = \hat{p}
\]
\[
\begin{align*}
\rho &= \text{hasField}(\hat{l}, \hat{p}, e_1, e_2) \\
\{m, p, \pi, x := \text{hasField}(e_1, e_2)\} &\xrightarrow{\text{cmd}} \{m', \hat{m}, \hat{p}[x \rightarrow \text{true}]\}
\end{align*}
\]

**METADATA**
\[
[e_1]_\rho = \hat{l}
\]
\[
\begin{align*}
\rho &= \text{metadata}(x, e) \\
\{m, p, \pi, x := \text{metadata}(e)\} &\xrightarrow{\text{cmd}} \{m, p[x \rightarrow \hat{v}]\}
\end{align*}
\]

**Property Assign 1:** When evaluating the command \([e_1, e_2] := e_3\), the semantics first evaluates the expressions \(e_1\) and \(e_2\), obtaining a location \(\hat{l}\) and property \(\hat{p}\) respectively obtaining the symbolic value \(\hat{v}\). The semantics will then search the heap for the value \(\hat{v}\) in the entry for \((\hat{l}, \hat{p})\). Then the path condition will be updated with the constraint \(\hat{l} \land \hat{p} \rightarrow \hat{v}\). Finally the value of \(x\) will be updated to be \(\hat{v}\).

**Property Assign 2:** When evaluating the command \([e_1, e_2] := e_3\), the semantics first evaluates the expressions \(e_1\), \(e_2\) and \(e_3\) to a location \(\hat{l}\), a property \(\hat{p}\) and a value \(\hat{v}\) respectively. Then the path condition will be updated with the constraint \(\hat{l} \land \hat{p} \rightarrow \hat{v}\). Finally the semantics finds in the heap an entry for \((\hat{l}, \hat{p})\) updating it to have value \(\hat{v}\).
Figure A.2: Symbolic Semantic of Information Flow Commands: $\{h, m, \tilde{p}, \pi, sc\} \Downarrow o$

Delete: When evaluating the command $\text{delete}(e_1, e_2)$, the semantics first evaluates the expressions $e_1$ and $e_2$ to a location $\tilde{l}$ and a property $\tilde{p}$ respectively. Then the path condition will be updated with the constraints $\tilde{l} = \tilde{l'} \land \tilde{p} = \tilde{p}'$. Finally the semantics finds an entry for $(\tilde{l}, \tilde{p})$, updating it by setting its value to be null.

Has Field - True: When evaluating the command $x := \text{hasField}(e_1, e_2)$, the semantics first evaluates the expressions $e_1$ and $e_2$ to a location $\tilde{l}$ and a property $\tilde{p}$ respectively. Then the path condition will be updated with the constraint $\tilde{l} = \tilde{l'} \land \tilde{p} = \tilde{p}'$. The Semantics finally finds an entry in the heap for $(\tilde{l}, \tilde{p})$, causing it to update the store value for $x$ to be true.

Has Field - false: When evaluating the command $x := \text{hasField}(e_1, e_2)$, the semantics first evaluates the expressions $e_1$ and $e_2$ to a location $\tilde{l}$ and a property $\tilde{p}$ respectively. Then the path condition will be updated with the constraint $\tilde{l} \neq \tilde{l'} \lor \tilde{p} \neq \tilde{p}'$. The Semantics does not find an entry in the heap for $(\tilde{l}, \tilde{p})$, causing it to update the store value for $x$ to be false.

Get Fields: When evaluating the command $x := \text{getFields}(e)$, the semantics first evaluates the expression $e$ to a location $\tilde{l}$. Then the path condition will be updated with the constraint $\tilde{l} = \tilde{l'}$. The Semantics finally searches the heap for all properties of $\tilde{l}$, updating the value of $\hat{x}$ with a list of the properties.

Metadata: When evaluating the command $x := \text{metadata}(e_1)$, the semantics first evaluates the expressions $e$ to a location $\tilde{l}$. Then the path condition will be updated with the constraint $\tilde{l} = \tilde{l'}$. The Semantics searches the metadata for a value $\hat{x}$ assigned to $\tilde{l}$. Finally $x$ is updated to have has its value $\hat{x}$.

A.1.2 Information Flow Commands

The semantics of symbolic JSIL information flow commands is given in Figure A.2 in a small-step style; semantic transitions have the form $\{h, m, \tilde{p}, \pi, sc\} \Downarrow o$, where:

- $\{h, m, \tilde{p}, \pi, sc\}$ is the initial configuration consisting of a symbolic heap, a symbolic metadata table, a symbolic store, a path condition (which is a formula that accumulates constraints on the symbolic variables that guide the execution to the current symbolic state) and a basic command to execute;

- $\Downarrow o$ is an event label to be given to the information flow monitor;

- $\{h, m, \tilde{p}\}$ is the final configuration consisting of the final symbolic heap, symbolic metadata and symbolic store.
Update Variable: The update variable \( \text{upgVar} (x, \sigma) \) semantics send an event to the monitor with the variable \( x \) and the level \( \sigma \).

Update Property: The update property \( \text{upgProp}(e_1, e_2, \sigma) \) semantics first evaluates \( e_1 \) and \( e_2 \) to a location \( \hat{i} \) and a property \( \hat{p} \) respectively. Then it updates the path condition \( \pi \) by adding the constraint \((\hat{i} = \hat{l}) \wedge (\hat{p} = \hat{p}')\). Now the semantics sends an event \( \text{upgProp}(\hat{l}', \hat{p}', e_1, e_2, \sigma) \) to the monitor.

Update Structure: The update structure \( \text{upgStruct}(e, \sigma) \) semantics first evaluates \( e \) to a location \( \hat{i} \). Then it updates the path condition \( \pi \) by adding the constraint \( \hat{i} = \hat{l}' \). Now the semantics sends an event \( \text{upgStruct}(\hat{l}', e, \sigma) \) to the monitor.

Update Metadata: The update structure \( \text{upgMtd}(e, \sigma) \) semantics first evaluates \( e \) to a location \( \hat{i} \). Then it updates the path condition \( \pi \) by adding the constraint \( \hat{i} = \hat{l}' \). Now the semantics sends an event \( \text{upgMtd}(\hat{l}', e, \sigma) \) to the monitor.

Merge: The merge \( \text{merge} (n) \) semantics sends an event \( \triangleright (n) \) to the monitor.

A.1.3 Control Flow Commands

The semantics of symbolic JSIL Control flow commands is given in Figure 3.2 in a small-step style; semantic transitions have the form \( \{\hat{h}, \hat{m}, \hat{p}, \pi, \hat{cs}, i\} \xrightarrow{\text{cmd}} \{\hat{h}', \hat{m}', \hat{p}', \pi, \hat{cs}', j\} \), where:

- \( \{\hat{h}, \hat{m}, \hat{p}, \pi, \hat{cs}, i\} \) is the initial configuration consisting of a symbolic heap, a symbolic metadata table, a symbolic store, a path condition, a symbolic call stack and the current index of the program;

- \( o \) is an event label to be given to symbolic the information flow monitor;

- \( \{\hat{h}', \hat{m}', \hat{p}', \pi, \hat{cs}, j\} \) is the final configuration consisting of the final symbolic heap, a symbolic metadata, a symbolic store, a path condition, a symbolic call stack and the index of the next command.

Below we give a detailed explanation of the semantics of each command.
**Basic Command:** The control flow semantics executes a basic command using the semantics of the basic command.

**Security Command:** The control flow semantics executes a Security command using the semantics of the Security command.

**Unconditional Goto:** When executing an unconditional goto goto $j$, the control flow semantics simply jumps to the $j$-th command of the current procedure.

**Conditional Goto:** When executing a conditional goto goto $[e] j$, the semantics first evaluates the guard expression $e$ to $\hat{v}$. Then the path condition $\pi$ is updated with the value of $\hat{v}$. If $\hat{v}$ is true, then the program will jump to the $j$-th command of the current procedure; otherwise it will jump to the $k$-th command of the current procedure.

**Procedure Call:** When executing a procedure call $x := e(e_k | n_k = 0)$ with $j$, the semantics first evaluates the caller expression $e$, and the arguments expressions $e_k | n_k = 0$, obtaining $\hat{v}_k | n_k = 1$. Then, it creates a new symbolic store $\hat{\rho}'$ for the execution of $f$. Finally, the semantics creates a new symbolic call stack entry to keep track of the execution context of the current function, so that control can correctly be returned once the execution of $f$ finishes.

**Normal Return:** When executing a normal return command, the program will search the current store for the value $v$ assigned to $ret$. The program will then discard the last entry of the call stack and update the store by assigning the value $v$ to $ret$.

**Throw:** When executing a throw command, the program will search the current store for the value $v$ assigned to $ret$. The program will then discard the last entry of the call stack and update the store by assigning the value $v$ to $ret$. 

81